A comparison of R tools for nonlinear least squares modeling

John C. Nash

Arkajyoti Bhattacharjee

2023-3-2

## Abstract

Our Google Summer of Code project “Improvements to nls()” considered the features and limitations of the R function nls() with the aim of improving and rationalizing R tools for nonlinear regression. The rich features of nls() are weakened by several deficiencies and inconsistencies such as a lack of stabilization of the Gauss-Newton solver. Further considerations are the usability and maintainability of the code base that provides the functionality nls() claims to offer. Various packages, including our nlsr, provide alternative capabilities. We consider the differences in goals, approaches, and features of different tools for nonlinear least squares modeling in R. Discussion of these matters is relevant to improving R generally as well as its nonlinear estimation tools.

## The nls() function

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

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

* handles formulas that include R functions, even ones that call calculations in other programming languages
* allows data to be weighted or subset
* can estimate bound-constrained parameters
* provides a mechanism for handling partially linear models
* permits parameters to be indexed over a set of related data
* produces measures of variability (i.e., standard error estimates) for the estimated parameters
* has related profiling capabilities for exploring the likelihood surface as parameters are changed
* links to many pre-coded (selfStart) models that do not require initial parameter values.

With such a range of features and long history, the code has become untidy and overly patched, difficult to maintain, and its underlying methods could be improved. Various workers have developed packages to overcome these concerns, and we will address some of these here.

## Scope of our comparison

The tools we will consider are the base-R nls() function and some packages that are available on the CRAN repository. Of these, we will pay particular attention to **nlsr** (John C Nash and Duncan Murdoch ([2023](#ref-nlsr2023manual))), **minpack.lm** (Elzhov et al. ([2012](#ref-minpacklm12))), and **gslnls** (Chau ([2023](#ref-gslnls23))) which are general nonlinear least-squares solvers.

While we will provide capsule comments for some other CRAN packages, we will not pursue those in the Bioconductor (Gentleman et al. ([2004](#ref-Bioconductor))) collection, nor those on repositories such as GitHub (<https://github.com>) and Gitlab (<https://about.gitlab.com>).

## Some other CRAN packages for nonlinear modeling

onls (Andrej-Nikolai Spiess ([2022](#ref-R-onls-manual))) – The usual optimization in estimating nonlinear models is the vertical difference between the dependent variable and the functional model. onls minimizes the sum of **orthogonal** residuals. The objective is therefore different and involves nontrivial extra calculation. A vignette with the package and the blog article <https://www.r-bloggers.com/2015/01/introducing-orthogonal-nonlinear-least-squares-regression-in-r/> give some description with illustrative graphs. onls appears to be limited to problems with one independent and one dependent variable. The Wikipedia article <https://en.wikipedia.org/wiki/Total_least_squares> presents an overview of some ideas, with references to the literature. The approach needs a wider discussion and tutorial examples to allow its merits to be judged than can be included here.

crsnls (Tvrdík ([2016](#ref-R-crsnls-manual))) – This package allows nonlinear estimation by controlled random search via two methods. There is unfortunately no vignette. A modest trial we carried out showed nlsr::nlxb() gave the same results in a small fraction of the time required by either of the methods in crsnls. The method discussed in Josef Tvrdík and Ivan Křivý and Ladislav Mišík ([2007](#ref-tvrdik2007)) claims better reliability in finding solutions than a Levenberg-Marquardt code (actually from Matlab), but the tests were conducted on the extreme NIST examples mentioned next.

NISTnls (National Institutes for Standards and R port by Douglas Bates ([2012](#ref-NISTnls-manual))) – This package provides R code and data for a set of (numerically ill-conditioned) nonlinear least squares problems from the U.S. National Institute for Standards and Technology. These may not represent real-world situations.

nlshelper (Duursma ([2017](#ref-nlshelper-manual))) – This package, which unfortunately lacks a vignette, provides a few utilities for summarizing, testing, and plotting non-linear regression models estimated with nls(), nlsList() or nlme() that are linked or grouped in some way.

nlsic (Sokol ([2022](#ref-nlsic-manual))) – This solves nonlinear least squares problems with optional equality and/or inequality constraints. It is clearly **not** about modeling, and the input and output are quite different from class nls methods. However, there do not appear to be other R packages with these capabilities.

nlsMicrobio (Baty and Delignette-Muller ([2014](#ref-nlsMicrobio-manual))) – Data sets and nonlinear regression models dedicated to predictive microbiology, including a vignette, by authors of the nlstools package.

nlstools (Baty and Delignette-Muller ([2013](#ref-p-nlstools))) – This package provides several tools for aiding the estimation of nonlinear models, particularly using nls(). The vignette is actually a journal article, and the authors have considerable experience in the subject.

nlsmsn (Prates, Lachos, and Garay ([2021](#ref-m-nlsmsn))) – Fit univariate non-linear scale mixture of skew-normal(NL-SMSN) regression, with details in Garay, Lachos, and Abanto-Valle (2011). The problem here is to minimize an objective that is modified from the traditional sum of squared residuals.

nls.multstart (Padfield and Matheson ([2020](#ref-nlsmultstart))) – Non-linear least squares regression using AIC scores with the Levenberg-Marquardt algorithm using multiple starting values for increasing the chance that the minimum found is the global minimum.

nls2 (Grothendieck ([2022](#ref-nls2-22))) – Nonlinear least squares by brute force has similar motivations to nls.multstart, but uses nls() within multiple trials. The author has extensive expertise in R.

nlstac (Rodriguez-Arias et al. ([2020](#ref-m-nlstac))) – A set of functions implementing the algorithm described in Fernandez Torvisco et al. (2018) for fitting separable nonlinear regression curves. The special class of problem for which this package is intended is an important and difficult one. No vignette is provided, unfortunately.

easynls – Fit and plot some nonlinear models. Thirteen models are treated, but there is minimal documentation and no vignette. Package nlraa is to be preferred.

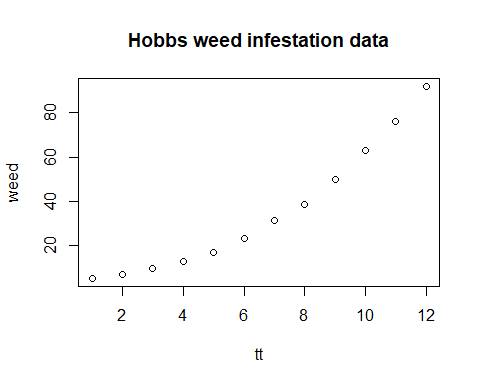
nlraa (Miguez ([2021](#ref-MiguezNLRAA2021))) – a set of nonlinear *selfStart* models, primarily from agriculture. Most include analytic Jacobian code.

optimx (John C. Nash and Varadhan ([2011](#ref-p-optimx))) – This provides optimizers that can be applied to minimize a nonlinear function which could be a nonlinear sum of squares. NOT generally recommended if nonlinear least squares programs can be easily used, but provides a check and alternative solvers.

## An illustrative example

The Hobbs weed infestation problem (John C. Nash ([1979, 120](#ref-cnm79))) is a growth curve modeling task that seems straightforward but is quite nasty. Its very succinct statement provides the “short reproducible example” much requested on R mailing lists. This is a real problem from a field researcher. The data and graph follow.

weed <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,  
 38.558, 50.156, 62.948, 75.995, 91.972)  
tt <- 1:12  
weeddf <- data.frame(tt, weed)  
plot(weeddf, main="Hobbs weed infestation data")



Three suggested models for this data are (with names to allow for easy reference)

**Logistic3U**:

**Logistic3S**:

**Logistic3T**:

where we will use weed for and tt for . The functions above are equivalent, but the first is generally more awkward to solve numerically due to its poor scaling. The parameters of the three forms are related as follows:

To allow for simpler discussion, let us say that the parameters form a (named) vector and the model function is called . The residuals can be written either as or since their sum of squares has an identical value. The second form avoids a potential sign error if we need to evaluate derivatives.

We wish to minimize the sum of squared residuals, which is our loss (or objective) function. Starting with some guess for the parameters, we aim to alter these parameters to obtain a smaller loss function. We then iterate until we can make no further progress.

Let us consider there are parameters and residuals. The loss function is

The gradient of is

where the Jacobian is given by elements

and the Hessian is defined by elements

If we expand the Hessian for nonlinear least squares problems, we find

Let us use for the elements of the second term of this expression. What is generally called **Newton’s method** for function minimization tries to set the gradient to zero (to find a stationary point of the function ). This leads to **Newton’s equation**

Given a set of parameters , we solve this equation for , adjust to and iterate, hopefully to converge on a solution. Applying this to a sum of squares problem gives

In this expression, only the elements of have second partial derivatives. Gauss, attempting to model planetary orbits, had small residuals, and noted that these multiplied the second partial derivatives of , so he approximated

by assuming . This results in the Gauss-Newton method where we solve

though we can avoid some loss of accuracy by NOT forming the inner product matrix and solving the linear least squares matrix problem

by one of several matrix decomposition methods.

In reality, there are many problems were should not be ignored, but the work to compute it precisely is considerable. Many work-arounds have been proposed, of which the Levenberg-Marquardt stabilization (Levenberg ([1944](#ref-Levenberg1944)), Marquardt ([1963](#ref-Marquardt1963))) is the most commonly used. For convenience, we will use “Marquardt”, as we believe he first incorporated the ideas into a practical computer program.

The usual suggestion is that be replaced by a multiple of the unit matrix or else a multiple of the diagonal part of . In low precision, some elements of could underflow to zero (John C. Nash ([1977](#ref-jn77ima))), and a linear combination of both choices is an effective compromise. Various choices for , as well as a possible line search along the direction rather than a unit step (Hartley ([1961](#ref-Hartley1961))), give rise to several variant algorithms. “Marquardt’s method” is a family of methods. Fortunately, most choices work well.

### Problem setup

Let us specify in R the three model formulas and set some starting values for parameters. These starting points are NOT equivalent and are deliberately crude choices. Workers performing many calculations of a similar nature should try to provide good starting points to reduce computation time and avoid finding a false solution.

# model formulas  
frmu <- weed ~ b1/(1+b2\*exp(-b3\*tt))  
frms <- weed ~ 100\*c1/(1+10\*c2\*exp(-0.1\*c3\*tt))  
frmt <- weed ~ Asym /(1 + exp((xmid-tt)/scal))  
#  
# Starting parameter sets  
stu1<-c(b1=1, b2=1, b3=1)  
sts1<-c(c1=1, c2=1, c3=1)  
stt1<-c(Asym=1, xmid=1, scal=1)

One of the useful features of nls() is the possibility of a selfStart model, where starting parameter values are not required. However, if a selfStart model is not available, nls() sets all the starting parameters to 1. This is tolerable, but could be improved by using a set of values that are all slightly different, which, in the case of the example model would avoid a singular Jacobian because and were equal in value. Program modifications to give a sequence like 1.0, 1.1, 1.2, 1.3 for the four parameters are fairly obvious.

It is also possible to provide R functions for the residual and Jacobian. This is usually more work for the user if the formula setup is possible. To illustrate, we show the functions for the unscaled 3 parameter logistic.

# Logistic3U  
hobbs.res <- function(x){ # scaled Hobbs weeds problem -- residual  
 # This variant uses looping  
 if(length(x) != 3) stop("hobbs.res -- parameter vector n!=3")  
 y <- 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  
 res <- x[1]/(1+x[2]\*exp(-x[3]\*tt)) - y  
}  
  
hobbs.jac <- function(x) { # scaled Hobbs weeds problem -- Jacobian  
 jj <- matrix(0.0, 12, 3)  
 tt <- 1:12  
 yy <- exp(-x[3]\*tt)  
 zz <- 1.0/(1+x[2]\*yy)  
 jj[tt,1] <- zz  
 jj[tt,2] <- -x[1]\*zz\*zz\*yy  
 jj[tt,3] <- x[1]\*zz\*zz\*yy\*x[2]\*tt  
 attr(jj, "gradient") <- jj  
 jj  
}

## Estimation of models specified as formulas

Using a formula specification was a principal advantage made with nls() when it became available in S sometime in the 1980s. It uses a Gauss-Newton (i.e., unstabilized) iteration with a step reduction line search. This works very efficiently as long as is not ill-conditioned. Below we see nls() does poorly on the example problem. To save page space, we use 1-line result display functions from package nlsr, namely pnls() and pshort().

unls1<-try(nls(formula=frmu, start=stu1, data=weeddf))

## Error in nls(formula = frmu, start = stu1, data = weeddf) :   
## singular gradient

snls1<-try(nls(formula=frms, start=sts1, data=weeddf))

## Error in nls(formula = frms, start = sts1, data = weeddf) :   
## singular gradient

tnls1<-try(nls(formula=frmt, start=stt1, data=weeddf))

## Error in nls(formula = frmt, start = stt1, data = weeddf) :   
## singular gradient

Here we see the infamous “singular gradient” termination message of nls().

### Solution attempts with nlsr

unlx1<-try(nlxb(formula=frmu, start=stu1, data=weeddf))  
print(unlx1)

## residual sumsquares = 2.5873 on 12 observations  
## after 19 Jacobian and 25 function evaluations  
## name coeff SE tstat pval gradient JSingval   
## b1 196.186 11.31 17.35 3.167e-08 -4.859e-09 1011   
## b2 49.0916 1.688 29.08 3.284e-10 -3.099e-08 0.4605   
## b3 0.31357 0.006863 45.69 5.768e-12 2.305e-06 0.04714

snlx1<-try(nlxb(formula=frms, start=sts1, data=weeddf))  
pshort(snlx1) # a short-form output

## snlx1 -- ss= 2.5873 : c1 = 1.9619 c2 = 4.9092 c3 = 3.1357; 34 res/ 23 jac

tnlx1<-try(nlxb(formula=frmt, start=stt1, data=weeddf))  
pshort(tnlx1) # alternatively print(tnlx1)

## tnlx1 -- ss= 2.5873 : Asym = 196.19 xmid = 12.417 scal = 3.1891; 36 res/ 27 jac

Though we have found solutions, the Jacobian is essentially singular as shown by its singular values. Note that these are **displayed** by package nlsr in a single column in the output to provide a compact layout, but the values do **NOT** correspond to the individual parameters in whose row they appear; they are a property of the whole problem.

### Solution attempts with minpack.lm

unlm1<-try(nlsLM(formula=frmu, start=stu1, data=weeddf))  
pnls(unlm1) # Short form of output

## unlm1 -- ss= 2.5873 : b1 = 196.19 b2 = 49.092 b3 = 0.31357; 17 itns

snlm1<-try(nlsLM(formula=frms, start=sts1, data=weeddf))  
pnls(snlm1)

## snlm1 -- ss= 2.5873 : c1 = 1.9619 c2 = 4.9092 c3 = 3.1357; 7 itns

tnlm1<-try(nlsLM(formula=frmt, start=stt1, data=weeddf))  
pnls(tnlm1) # short form to give sum of squares, else use summary(tnlm1)

## tnlm1 -- ss= 9205.4 : Asym = 35.532 xmid = 43376 scal = -2935.4; 39 itns

### Solution attempts with gslnls

library(gslnls)  
ugslnls1<-try(gsl\_nls(fn = frmu, data = weeddf, start = stu1))  
pnls(ugslnls1) # to get sum of squares

## ugslnls1 -- ss= 2.5873 : b1 = 196.19 b2 = 49.092 b3 = 0.31357; 25 itns

sgslnls1<-try(gsl\_nls(fn = frms, data = weeddf, start = sts1))  
pnls(sgslnls1) # Use summary() to get display

## sgslnls1 -- ss= 2.5873 : c1 = 1.9619 c2 = 4.9092 c3 = 3.1357; 9 itns

tgslnls1<-try(gsl\_nls(fn = frmt, data = weeddf, start = stt1))  
pnls(tgslnls1)

## tgslnls1 -- ss= 9205.4 : Asym = 35.532 xmid = 20847 scal = -1744.7; 47 itns

### Comparison notes for formula-setup solutions

nlsr::nlxb() uses print() to output standard errors and singular values of the Jacobian (for diagnostic purposes). By contrast, minpack.lm::nlsLM() and nls() use summary(), which does NOT display the sum of squares, while print() gives the sum of squares, but not the standard error of the residuals.

The singular values allow us to gauge how “nearly singular” the Jacobian is at the solution, and the ratio of the smallest to largest of the singular values is a simple but effective measure. The ratios are 4.6641e-05 for Logistic3U, 0.021022 for Logistic3S and 0.001055 for Logistic3T, so Logistic3S is the “least singular”.

The results from nlsLM and gsl\_nls for the transformed model Logistic3T have a very large sum of squares, which may suggest that these programs have failed. Since nls(), nlsLM(), and gsl\_nls() do not offer the singular values, we need to extract the Jacobian and compute its singular values. The following script shows how to do this, using as Jacobian what is called the gradient element in the returned solution for these solvers.

# for nlsLM  
if (inherits(tnlm1, "try-error")) {  
 print("Cannot compute solution -- likely singular Jacobian")  
 } else {   
 JtnlsLM <- tnlm1$m$gradient() # actually the Jacobian  
 svd(JtnlsLM)$d # Singular values  
}

## [1] 3.4641e+00 3.2530e-10 8.9643e-12

# for gsl\_nls  
if (inherits(tgslnls1, "try-error")) {  
 cat("Cannot compute solution -- likely singular Jacobian")  
} else {   
 JtnlsLM <- tgslnls1$m$gradient()  
 svd(JtnlsLM)$d # Singular values  
}

## [1] 3.4641e+00 1.0111e-08 4.7443e-11

We see that there are differences in detail, but the more important result is that two out of three singular values are essentially 0. Our Jacobian is singular, and no method of the Gauss-Newton type should be able to continue. Indeed, from the parameters reported at this saddle point, nlsr::nlxb() cannot proceed.

stspecial<- c(Asym = 35.532, xmid = 43376, scal = -2935.4)  
badstart<-nlxb(formula=frmt, start=stspecial, data=weeddf)  
print(badstart)

## residual sumsquares = 9205.4 on 12 observations  
## after 2 Jacobian and 2 function evaluations  
## name coeff SE tstat pval gradient JSingval   
## Asym 35.5321 NA NA NA -9.694e-09 3.464   
## xmid 43376 NA NA NA -1.742e-09 2.61e-10   
## scal -2935.4 NA NA NA -2.4e-08 7.12e-16

## Functional specification of problems

We illustrate how to solve nonlinear least squares problems using a function to define the residual. Note that gsl\_nls() requires a vector y that is the expected value of what we have called the residual, but in this case is actually the model. gsl\_nls uses a numerical approximation for the Jacobian if the argument jac is missing. Note function nlsr::pnlslm() for a 1-line display of the results of minpack.lm::nls.lm().

hobnlfb<-nlfb(start=stu1, resfn=hobbs.res, jacfn=hobbs.jac)  
pshort(hobnlfb) # use print(hobnlfb) for more detail

## hobnlfb -- ss= 2.5873 : b1 = 196.19 b2 = 49.092 b3 = 0.31357; 25 res/ 19 jac

hobnlm<-nls.lm(par=stu1, fn=hobbs.res, jac=hobbs.jac)  
pnlslm(hobnlm)

## hobnlm -- ss= 2.5873 : b1 = 196.19 b2 = 49.092 b3 = 0.31357; 17 itns

hobgsln<-gsl\_nls(start=stu1, fn=hobbs.res, y=rep(0,12))  
pnls(hobgsln)

## hobgsln -- ss= 2.5873 : b1 = 196.19 b2 = 49.092 b3 = 0.31357; 25 itns

hobgsl<-gsl\_nls(start=stu1, fn=hobbs.res, y=rep(0,12), jac=hobbs.jac)  
pnls(hobgsl) # using analytic Jacobian

## hobgsl -- ss= 2.5873 : b1 = 196.19 b2 = 49.092 b3 = 0.31357; 25 itns

## Design goals, termination tests and output objects

The output object of nlxb() is smaller than the class nls object returned by nls(), nlsLM() and gsl\_nls(). Package nlsr emphasizes the solution of the nonlinear least squares problem rather than the estimation of a nonlinear model that fits or explains the data. The object of class nls allows for a number of specialized modeling and diagnostic extensions. For compatibility, the nlsr package has function wrapnlsr(), for which nlsr() is an alias. This uses nlxb() to find good parameters, then calls nls() to return the class nls object. Unless particular modeling features are needed, the use of wrapnlsr() is unnecessary and wasteful of resources.

The design goals of the different tools may also be revealed in the so-called “convergence tests” for the iterative solvers. In the manual page for nls() in R 4.0.0 there was the warning:

*Do not use nls on artificial “zero-residual” data.*

with suggested addition of small perturbations to the data. This admits nls() could not solve well-posed problems unless data is polluted with errors. Zero-residual problems are not always artificial, since problems in function approximation and nonlinear equations can be approached with nonlinear least squares. Fortunately, a small adjustment to the “termination test” for the **program**, rather than for “convergence” of the underlying **algorithm**, fixes the defect. The test is the Relative Offset Convergence Criterion (see Bates, Douglas M. and Watts, Donald G. ([1981](#ref-BatesWatts81))). This scales an estimated reduction in the loss function by its current value. If the loss function is very small, we are close to a zero-divide. Adding a small quantity to the divisor avoids trouble. In 2021, one of us (J. Nash) proposed that nls.control() have an additional parameter scaleOffset with a default value of zero. Setting it to a small number – 1.0 is a reasonable choice – allows small-residual problems (i.e., near-exact fits) to be dealt with easily. We call this the **safeguarded relative offset convergence criterion**, and it has been in nlsr since it was introduced. The default value gives the legacy behaviour. This improvement has been in the R distributed code since version 4.1.0.

Additional termination tests can be used. nlsr has a **small sum of squares** test (**smallsstest**) that compares the latest evaluated sum of squared (weighted) residuals to e4 times the initial sum of squares, where e4 <- (100\*.Machine$double.eps)^4 is approximately 2.43e-55.

Termination after what may be considered excessive computation is also important. nls() stops after maxiter “iterations”. The meaning of “iteration” may require examination of the code. We terminate execution when the number of residual or jacobian evaluations exceed set limits. Generally we prefer larger limits than the default maxiter = 50 of nls() to avoid stopping early.

### Returned results of nls() and other tools

As mentioned, the output of nls() is an object of class “nls” which has a quite rich structure described in the manual file or revealed by applying the str() function to the result of nls().

The complexity of this object is a challenge to users. Let us use result as the returned object from nls(), minpack.lm::nlsLM() or gslnls::gsl\_nls(). For example, the data return element is an R symbol. To actually access the data from this element, we need to use the syntax:

eval(parse(text=result$data))

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

ls(result$model)

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

By contrast nlsr::nlxb() returns a much simpler structure of 11 items in one-level. Moreover, nlxb explicitly returns the sum of squares, the residual vector, Jacobian and counts of evaluations.

### When to compute ancillary information

Tools that produce a class nls output object create a rich set of functions and structures that are then used in a variety of modeling tasks, including the least squares solution. By contrast, nlsr computes quantities as they are requested or needed, with additional features in separate functions. For example, the singular values of the Jacobian are actually computed in the print and summary methods for the result. These two approaches lead to different consequences for performance and how features are provided. nlsr has antecendants in the methods of John C. Nash ([1979](#ref-cnm79)), where storage for data and programs was at a ridiculous premium in the small computers of the era. Thus the code in nlsr is likely of value for workers to copy and modify it for customized tools.

## Jacobian calculation

Gauss-Newton/Marquardt methods all need a Jacobian matrix at each iteration. By default, nlsr::nlxb() will try to evaluate this using analytic expressions using symbolic and automatic differentiation tools. When using a formula specification of the model, nls(), minpack.lm::nlsLM() and gslnls::gsl\_nls() use a finite difference approximation to compute the Jacobian, though gsl\_nls() does have an option to attempt symbolic expressions. Package nlsr provides, via appropriate calling syntax, four numeric approximation options for the Jacobian, with a further control ndstep for the size of the step used in the approximation.

Using the “gradient” attribute of the output of the Jacobian function to hold the Jacobian matrix lets us embed this in the **residual** function as well, so that the call to nlsr::nlfb() can be made with the same name used for both residual and Jacobian function arguments. This programming trick saves a lot of trouble for the package developer, but it can be a nuisance for users trying to understand the code.

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

numericDeriv() uses a simple forward difference approximation of derivatives, though a central difference approximation can be specified in control parameters. We are unclear why numericDeriv() in base R calls C\_numeric\_deriv, as we were easily able to create a more compact version entirely in R. See [*https://github.com/nashjc/RNonlinearLS/tree/main/DerivsNLS*](https://github.com/nashjc/RNonlinearLS/tree/main/DerivsNLS).

minpack.lm::nlsLM() invokes numericDeriv() in its local version of nlsModel(), but it appears to use an internal approximate Jacobian code from the original Fortran minpack code, namely, lmdif.f. Such differences in approach can lead to different behaviour, usually minor, but sometimes annoying with ill-conditioned problems.

* A pasture regrowth problem (Huet et al. ([2004](#ref-Huet2004)), page 1, based on Ratkowsky ([1983](#ref-Ratkowsky1983))) has a poorly conditioned Jacobian and nls() fails with “singular gradient”. Worse, numerical approximation to the Jacobian gives the error “singular gradient matrix at initial parameter estimates” for minpack.lm::nlsLM so that the Marquardt stabilization is unable to take effect, while the analytic derivatives of nlsr::nlxb give a solution.
* Karl Schilling (private communication) provided an example where a model specified with the formula y ~ a \* (x ˆ b) causes nlsr::nlxb to fail because the partial derivative w.r.t. b is a \* (x^b \* log(x)). If there is data for which x = 0, the derivative is undefined, but the model can be computed. In such cases, we observed that nls() and minpack.lm::nlsLM found a solution, though this seems to be a lucky accident.

### Jacobian code in selfStart models

Analytic Jacobian code can be provided to all the solvers discussed. Most selfStart models that automatically provide starting parameters also include such code. There is documentation in R of selfStart models, but their construction is non-trivial. A number of such models are included with base R in ./src/library/stats/R/zzModels.R, with package nlraa (Miguez ([2021](#ref-MiguezNLRAA2021))) providing a richer set. There are also some in the now-archived package NRAIA. These provide the Jacobian in the “gradient” attribute of the “one-sided” formula that defines each model, and these Jacobians are often the analytic forms.

The nls() function, after computing the “right hand side” or rhs of the residual, checks to see if the “gradient” attribute is defined, otherwise using numericDeriv() to compute a Jacobian into that attribute. This code is within the nlsModel() or nlsModel.plinear() functions. The use of analytic Jacobians almost certainly contributes to the good performance of nls() on selfStart models.

The use of selfStart models with nlsr is described in the “Introduction to nlsr” vignette. However, since nlsr generally can use very crude starting values, we have rarely needed them, though it should be pointed out that our work is primarily diagnostic. If we were carrying out a large number of similar estimations, such initial parameters are critical to efficiency.

In considering selfStart models, we noted that the base-R function SSlogis is intended to solve problem **Logistic3T** above. When this function is used via getInitial() to find starting values, it actually calls nls() with the ‘plinear’ algorithm and finds a (full) solution. It then passes the solution coefficients to the default algorithm unnecessarily. Moreover, the implicit double call is, in our view, prone to creating errors in code maintenance. To provide simpler starting parameters, the function SSlogisJN is now part of package nlsr, but is most useful for nls().

Users may also want to profit from the Jacobian code of selfStart models but supply explicit starting values other than those suggested by getInitial(). This does not appear to be possible with nls(). nlsr::nlxb() always requires starting parameters, and can either use getInitial() to find them from the selfStart model or provide explicit values, but the selfStart model is used in the formula for the nlxb() call.

We are also surprised that the analytic expressions for the Jacobian (“gradient”) in the SSLogis function and others save quantities in “hidden” variables, i.e., with names preceded by “.”. These are then not displayed by the ls() command, making them difficult to access by users who may wish to create their own selfStart model via copy and edit. Interactive tools, such as “visual fitting” (John C. Nash and Velleman ([1996](#ref-nash1996nonlinear))) might be worth considering as another way to find starting parameters, but we know of no R capability of this type.

As a side note, the introduction of scaleOffset in R 4.1.1 to deal with the convergence test for small residual problems now requires that the getInitial() function have dot-arguments (...) in its argument list. This illustrates the entanglement of many features in nls() that complicate its maintenance and improvement.

## Bounds constraints on parameters

For many problems we know that parameters cannot be bigger or smaller than some externally known limits. Such limits should be built into models, but there are some important details for using the tools in R.

* nls() can only impose bounds if the algorithm="port" argument is used in the call. Unfortunately, the documentation warns us:
* *The algorithm = “port” code appears unfinished, and does not even check that the starting value is within the bounds. Use with caution, especially where bounds are supplied.*
* gsl\_nls() does not offer bounds.
* bounds are part of the default method for package nlsr.
* nlsLM() includes bounds in the standard call, but we have observed cases where it fails to get the correct answer. From examination of the code, we believe the authors have not taken into account all possibilities, though all programs have some weakness regarding constrained optimization in that programmers have to work with assumptions on the scale of numbers, and some problems will be outside the scope envisaged.

# Start MUST be feasible i.e. on or within bounds  
anlshob1b <- nls(frms, start=sts1, data=weeddf, lower=c(0,0,0),  
 upper=c(2,6,3), algorithm='port')  
pnls(anlshob1b) # check the answer (short form)

## anlshob1b -- ss= 9.4726 : c1 = 2 c2 = 4.4332 c3 = 3; 10 itns

# nlsLM seems NOT to work with bounds in this example  
anlsLM1b <- nlsLM(frms, start=sts1, data=weeddf, lower=c(0,0,0), upper=c(2,6,3))  
pnls(anlsLM1b)

## anlsLM1b -- ss= 881.02 : c1 = 2 c2 = 6 c3 = 3; 2 itns

# also no warning if starting out of bounds, but gets good answer!!  
st4<-c(c1=4, c2=4, c3=4)  
anlsLMob <- nlsLM(frms, start=st4, data=weeddf, lower=c(0,0,0), upper=c(2,6,3))  
pnls(anlsLMob)

## anlsLMob -- ss= 9.4726 : c1 = 2 c2 = 4.4332 c3 = 3; 4 itns

# Try nlsr::nlxb()  
anlx1b <- nlxb(frms, start=sts1, data=weeddf, lower=c(0,0,0), upper=c(2,6,3))  
pshort(anlx1b)

## anlx1b -- ss= 9.4726 : c1 = 2 c2 = 4.4332 c3 = 3; 12 res/ 12 jac

### Philosophical considerations

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

### Fixed parameters (masks)

Let us try to fix (mask) the first parameter in the first two example problems.

# Hobbsmaskx.R -- masks with formula specification of problem  
require(nlsr); require(minpack.lm); traceval<-FALSE  
stu <- c(b1=200, b2=50, b3=0.3) # a default starting vector (named!)  
sts <- c(c1=2, c2=5, c3=3) # a default scaled starting vector (named!)  
# fix first parameter  
anxbmsk1 <- try(nlxb(frmu, start=stu, data=weeddf, lower=c(200,0,0),   
 upper=c(200, 60, 3), trace=traceval))  
print(anxbmsk1)

## residual sumsquares = 2.6182 on 12 observations  
## after 4 Jacobian and 4 function evaluations  
## name coeff SE tstat pval gradient JSingval   
## b1 200U M NA NA NA 0 NA   
## b2 49.5108 1.12 44.21 8.421e-13 -2.887e-07 1022   
## b3 0.311461 0.002278 136.8 1.073e-17 0.0001635 0.4569

anlM1 <- try(nlsLM(frmu, start=stu, data=weeddf, lower=c(200,0,0),   
 upper=c(200, 60, 3), trace=traceval))  
pnls(anlM1)

## anlM1 -- ss= 2.6182 : b1 = 200 b2 = 49.511 b3 = 0.31146; 4 itns

anlsmsk1 <- try(nls(frmu, start=stu, data=weeddf, lower=c(200,0,0),   
 upper=c(200, 60, 3), algorithm="port", trace=traceval))  
pnls(anlsmsk1)

## anlsmsk1 -- ss= 2.6182 : b1 = 200 b2 = 49.511 b3 = 0.31146; 5 itns

# Hobbs scaled problem with bounds, formula specification  
anlxmsks1 <- nlxb(frms, start=sts, data=weeddf, lower=c(2,0,0),  
 upper=c(2,6,30))  
print(anlxmsks1)

## residual sumsquares = 2.6182 on 12 observations  
## after 4 Jacobian and 4 function evaluations  
## name coeff SE tstat pval gradient JSingval   
## c1 2U M NA NA NA 0 NA   
## c2 4.95108 0.112 44.21 8.421e-13 -2.981e-06 104.2   
## c3 3.11461 0.02278 136.8 1.073e-17 1.583e-05 4.482

anlshmsk1 <- nls(frms, start=sts, trace=traceval, data=weeddf, lower=c(2,0,0),  
 upper=c(2,6,30), algorithm='port')  
pnls(anlshmsk1)

## anlshmsk1 -- ss= 2.6182 : c1 = 2 c2 = 4.9511 c3 = 3.1146; 5 itns

anlsLMmsks1 <- nlsLM(frms, start=sts, data=weeddf, lower=c(2,0,0),  
 upper=c(2,6,30))  
pnls(anlsLMmsks1)

## anlsLMmsks1 -- ss= 2.6182 : c1 = 2 c2 = 4.9511 c3 = 3.1146; 4 itns

# Test with all parameters masked  
anlxmskall<- try(nlxb(frms, start=sts, data=weeddf, lower=sts, upper=sts))

## Warning in nlfb(start = pnum, resfn = trjfn, jacfn = trjfn, trace = trace, :  
## All parameters are masked

print(anlxmskall)

## Warning in summary.nlsr(x): No unmasked parameters

## residual sumsquares = 158.23 on 12 observations  
## after 0 Jacobian and 1 function evaluations  
## name coeff SE tstat pval gradient JSingval   
## c1 2U M NA NA NA NA NA   
## c2 5U M NA NA NA NA NA   
## c3 3U M NA NA NA NA NA

nlsr has an output format that indicates the constraint status of the parameter estimates. For nlsr we have **chosen** to suppress calculation of approximate standard errors in the parameters when constraints are active because their meaning under constraints is unclear, though we believe this policy worthy of discussion and further investigation.

## Stabilization of Gauss-Newton computations

All four major tools illustrated solve some variant of the Gauss-Newton equations. nls() uses a modification of an approach suggested by Hartley ([1961](#ref-Hartley1961)), while nlsr, gslnls and minpack.lm use flavours of Marquardt ([1963](#ref-Marquardt1963)). gslnls offers an accelerated Marquardt method and three alternative methods which we have not investigated. Control settings for nlxb() or nlfb() allow exploration of Hartley and Marquardt algorithm variants. In general, the Levenberg-Marquardt stabilization is important in obtaining solutions in methods of the Gauss-Newton family, as nls() terminates too frequently and unnecessarily with singular gradient errors.

### Programming language

An important choice made in developing nlsr was to code entirely within the R programming language. nls() uses a mix of R, C and Fortran, as does minpack.lm. gslnls is an R wrapper to various C-language routines in the Gnu Scientific Library (Galassi et al. ([2009](#ref-GSL-manual))). Generally, keeping to a single programming language can allow for easier maintenance and upgrades. It also avoids some work when there are changes or upgrades to libraries for the non-R languages. R is usually considered slower than most computing environments because it keeps track of objects and because it is usually interpreted. In recent years the performance penalty for using code entirely in R has been much reduced with the just-in-time compiler and other improvements. All-R computation may now offer acceptable performance. In nlsr the use of R may be a smaller performance cost than the aggressive approach to a solution, which can cause more iterations to be used.

## Data sources to problems

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

### Feature: Subsetting

nls() and other class nls tools accept an argument subset. This acts through the mediation of model.frame, which not obvious in the source code files /src/library/stats/R/nls.R and /src/library/stats/src/nls.C. Having subset at the level of the call to a function like nls() saves effort, but it does mean that the programmer of the solver needs to be aware of the origin (and value) of objects such as the data, residuals and Jacobian. By preference, we would implement subsetting by zero-value weights, with observation counts (and degrees of freedom) computed via the numbers of non-zero weights. Alternatively, we would extract a working dataframe from the relevant elements in the original.

### Feature: na.action (missing value treatment)

na.action is an argument to the nls() function, but it does not appear obviously in the source code, often being handled behind the scenes after referencing the option na.action. This feature also changes the data supplied to our nonlinear least squares solver. A useful, but possibly dated, description is given in: [*https://stats.idre.ucla.edu/r/faq/how-does-r-handle-missing-values/*](https://stats.idre.ucla.edu/r/faq/how-does-r-handle-missing-values/). The typical default action, which can be seen by using the command getOption("na.action") is na.omit. This option removes from computations any observations containing missing values (i.e. any row of a data frame containing an NA). na.exclude does much of the same for solver computations, but keeps the rows with NA elements so that predictions are in the correct row position. We recommend that workers actually test output to verify the behaviour is as wanted. See [*https://stats.stackexchange.com/questions/492955/should-i-use-na-omit-or-na-exclude-in-a-linear-model-in-r*](https://stats.stackexchange.com/questions/492955/should-i-use-na-omit-or-na-exclude-in-a-linear-model-in-r). As with subset, our concern with na.action is that users may be unaware of the effects of an option they may not even be aware has been set. Should na.fail be the default?

## Feature: model frame

model is an argument to the nls() and related functions, which is documented as:

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

Indeed, the argument only gets used when nls() is about to return its result object, and the element model is NULL unless the calling argument model is TRUE. (Using the same name could be confusing.) Despite this, the model frame is used within the function code in the form of the object mf. We feel that users could benefit from more extensive documentation and examples of its use since it is used to implement features like subset.

## Weights on observations

All four main tools we consider here allow a weights argument that specifies a vector of fixed weights the same length as the number of residuals. Each residual is multiplied by the square root of the corresponding weight. Where available, the values returned by the residuals() function are weighted, and the fitted() or predict() function are used to compute raw residuals.

While fixed weights may be useful, there are many problems for which we want weights that are determined at least partially from the model parameters, for example, a measure of the standard deviation of observations. Such dynamic weighting situations are discussed in the vignette “Introduction to nlsr” of package nlsr in section *Weights that are functions of the model parameters*. minpack.lm offers a function wfct() to facilitate such weighting. Care is advised in applying such ideas.

### Weights in returned functions from nls()

The function resid() (an alias for residuals()) gives WEIGHTED residuals, as does, for example, result$m$resid(). The function nlsModel(), which we have had to extract from the base R code and explicitly source() because it is not exported to the working namespace, allows us to compute residuals for particular coefficient sets.

wts <- 0.5^tt # simple weights  
frmlogis <- weed ~ Asym/(1 + exp((xmid - tt)/scal))  
Asym<-1; xmid<-1; scal<-1  
nowt<-nls(weed ~ SSlogis(tt, Asym, xmid, scal)) # UNWEIGHTED  
rnowt<-nowt$m$resid() # This has UNWEIGHTED residual and Jacobian. Does NOT take coefficients.  
attr(rnowt, "gradient")<-NULL; rnowt

## [1] -0.011900 0.032755 -0.092030 -0.208782 -0.392634 0.057594 1.105728  
## [8] -0.715786 0.107647 0.348396 -0.652592 0.287569

usewt <- nls(weed ~ SSlogis(tt, Asym, xmid, scal), weights=wts)  
rusewt<-usewt$m$resid() # WEIGHTED. Does NOT take coefficients.  
attr(rusewt,"gradient")<-NULL; rusewt

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

source("nlsModel.R")  
nmod0 <- nlsModel(frmlogis, data=weeddf, start=c(Asym=1, xmid=1, scal=1), wts=wts)  
rn0<-nmod0$resid() # Parameters are supplied in nlsModel() `start` above.  
attr(rn0,"gradient")<-NULL; rn0

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

nmod <- nlsModel(frmlogis, data=weeddf, start=coef(usewt), wts=wts)  
rn<-nmod$resid()  
attr(rn,"gradient")<-NULL; rn

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

## Minor issues with nonlinear least-squares tools

### Interim output from the “port” algorithm

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

### Failure to return the best result achieved

If nls() reaches a point where it cannot continue but has not found a point where the relative offset convergence criterion is met, it may simply exit, especially if a “singular gradient” (singular Jacobian) is found. However, this may occur AFTER the function has made considerable progress in reducing the sum of squared residuals. Here is an abbreviated example:

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

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

print(tryit)

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

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

## Estimating models that are partially linear

The variable projection method (Golub and Pereyra ([1973](#ref-Golub1973)), O’Leary and Rust ([2013](#ref-OlearyRust13))) is generally much more effective than general approaches in finding good solutions to nonlinear least squares problems when some of the parameters appear linearly. In our logistic examples, the asymptote parameters are an illustration. However, identifying which parameters are linear and communicating this information to estimating functions is not a trivial task. nls() has an option algorithm="plinear" that allows some partially linear models to be solved. The other tools, as far as we are aware, do not offer any such capability. The nlstac package uses a different algorithm for similar goals.

Within nls() itself we must, unfortunately, use different specifications with different algorithm options. For example, the explicit model y ~ a \* x + b does not work with the linear modeling function lm(), which requires this model to be specified as y ~ x. Within nls(), consider the following FOUR different specifications for the same problem, plus an intuitive choice, labelled fm2a, that does NOT work. In this failed attempt, putting the Asym parameter in the model causes the plinear algorithm to try to add another term to the model. We believe this is unfortunate, and would like to see a consistent syntax. At the time of writing, we do not foresee a resolution for this issue. In the example, we have NOT evaluated the commands to save space.

DNase1 <- subset(DNase, Run == 1) # select the data  
## using a selfStart model - do not specify the starting parameters  
fm1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)  
summary(fm1)  
  
## using conditional linearity - leave out the Asym parameter  
fm2 <- nls(density ~ 1 / (1 + exp((xmid - log(conc)) / scal)),  
 data = DNase1, start = list(xmid = 0, scal = 1),  
 algorithm = "plinear")  
summary(fm2)  
  
## without conditional linearity  
fm3 <- nls(density ~ Asym / (1 + exp((xmid - log(conc)) / scal)),  
 data = DNase1,  
 start = list(Asym = 3, xmid = 0, scal = 1))  
summary(fm3)  
  
## using Port's nl2sol algorithm  
fm4 <- try(nls(density ~ Asym / (1 + exp((xmid - log(conc)) / scal)),  
 data = DNase1, start = list(Asym = 3, xmid = 0, scal = 1),  
 algorithm = "port"))  
summary(fm4)  
  
## using conditional linearity AND Asym does NOT work  
fm2a <- try(nls(density ~ Asym / (1 + exp((xmid - log(conc)) / scal)),   
 data = DNase1, start = list(Asym=3, xmid = 0, scal = 1),  
 algorithm = "plinear", trace = TRUE))  
summary(fm2a)

## Models with indexed parameters

Some models have several common parameters and others that are tied to particular cases. The **man** file for nls() includes an example of a situation in which parameters are indexed but which uses the “plinear” option as an added complication. Running this example reveals that the answers for the parameters are NOT indexed as in a vector. That is, we do not see a[1], a[2], a[3] but a1, a2, a3. This is no doubt because programming for indexed parameters is challenging. We note that there are capabilities in packages for mixed-effects modelling such as nlme (Pinheiro et al. ([2013](#ref-p-nlme))), bbmle (Bolker and Team ([2013](#ref-p-bbmle))), and lme4 (p-lme4) for estimating models of such type.

## Tests and use-case examples

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

* to ensure packages work properly, in particular, giving results comparable to or better than the functions they are to replace.
* to test individual solver functions to ensure they work across the range of calling mechanisms, that is, different ways of supplying inputs to the solver(s);
* to pose “silly” inputs to see if these bad inputs are caught by the programs.

Such goals align with the aims of **unit testing** (e.g., [*https://towardsdatascience.com/unit-testing-in-r-68ab9cc8d211*](https://towardsdatascience.com/unit-testing-in-r-68ab9cc8d211), Wickham ([2011](#ref-HWtestthat11)), Wickham et al. ([2021](#ref-HWdevtools21)) and the conventional R package testing tools). In our work, one of us has developed a working prototype package at [*https://github.com/ArkaB-DS/nlsCompare*](https://github.com/ArkaB-DS/nlsCompare) . A primary design objective of this is to allow the summarisation of multiple tests in a compact output. The prototype has a vignette to illustrate its use.

## Documentation and resources

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

* A BibTex bibliography for use with all documents in this project, but which has wider application to nonlinear least squares projects in general ([*https://github.com/nashjc/RNonlinearLS/blob/main/BibSupport/ImproveNLS.bib*](https://github.com/nashjc/RNonlinearLS/blob/main/BibSupport/ImproveNLS.bib)).
* MachID.R offers a suggested concise summary function to identify a particular computational system used for tests. A discussion of how it was built and the resources needed across platforms is given in at [*https://github.com/nashjc/RNonlinearLS/tree/main/MachineSummary*](https://github.com/nashjc/RNonlinearLS/tree/main/MachineSummary).
* John C. Nash and Bhattacharjee ([2022](#ref-PkgFromRbase22)) is an explanation of the construction of the nlspkg from the nls() code in R-base.
* As the 2021 Summer of Code period was ending, one of us (JN) was invited to prepare a review of optimization in R. Ideas from the present work have been instrumental in the creation of John C. Nash ([2022](#ref-NashWires22)).

## Future of nonlinear model estimation in R

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

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

* R tools presume data and parameters needed are available in an accessible environment. This provides a compact syntax to invoke the calculations, but the wrong data can be used if the internal search finds a valid name that is not the object we want, or if subset or na.action settings modify the selection, or weights are applied.
* Mixing of R, C and Fortran code adds to the burden of following the program logic.
* The class nls structure simplifies calls with its rich set of functions, but also adds to the task of understanding what has been done. A design that isolates the setup, solution, and post-solution parts of complicated calculations reduces the number of objects that must be kept in alignment.

Given the existence of examples of good practices such as analytic derivatives, stabilized solution of Gauss-Newton equations and bounds-constrained parameters, base R tools should be moving to incorporate them. These capabilities are available now by using several tools, but it would be helpful if they were unified.

# Acknowledgement

Hans Werner Borchers was helpful in developing GSoC project motivating this article and in comments on this and related work. Heather Turner co-mentored the project and helped guide the progress of the work. Exchanges with Fernando Miguez helped to clarify aspects of selfStart models and instigated the “Introduction to nlsr” vignette. Colin Gillespie (package benchmarkme) has been helpful in guiding our attempts to succinctly summarize computing environments.

# References

Andrej-Nikolai Spiess. 2022. *Onls: Orthogonal Nonlinear Least-Squares Regression*. <https://CRAN.R-project.org/package=onls>.

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

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

Baty, Florent, and Marie-Laure Delignette-Muller. 2013. *Nlstools: Tools for Nonlinear Regression Diagnostics*.

———. 2014. *nlsMicrobio: Data Sets and Nonlinear Regression Models Dedicated to Predictive Microbiology*.

Bolker, Ben, and R Development Core Team. 2013. *Bbmle: Tools for General Maximum Likelihood Estimation*. <http://CRAN.R-project.org/package=bbmle>.

Chau, Joris. 2023. *Gslnls: GSL Nonlinear Least-Squares Fitting*. <https://CRAN.R-project.org/package=gslnls>.

Duursma, Remko. 2017. *Nlshelper: Convenient Functions for Non-Linear Regression*. <https://CRAN.R-project.org/package=nlshelper>.

Elzhov, Timur V., Katharine M. Mullen, Andrej-Nikolai Spiess, and Ben Bolker. 2012. *Minpack.lm: R Interface to the Levenberg-Marquardt Nonlinear Least-Squares Algorithm Found in MINPACK, Plus Support for Bounds*. R Project for Statistical Computing. <http://CRAN.R-project.org/package=minpack.lm>.

Galassi, Mark, Jim Davies, James Theiler, Brian Gough, and Gerard Jungman. 2009. *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12 (3. Ed.).*

Gentleman, Robert C., Vincent J. Carey, Douglas M. Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, et al. 2004. “Bioconductor: Open Software Development for Computational Biology and Bioinformatics.” *Genome Biology* 5 (R80). <https://doi.org/10.1186/gb-2004-5-10-r80>.

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

Grothendieck, G. 2022. *Nls2: Non-Linear Regression with Brute Force*. <https://CRAN.R-project.org/package=nls2>.

Hartley, H. O. 1961. “The Modified Gauss-Newton Method for the Fitting of Non-Linear Regression Functions by Least Squares.” *Technometrics* 3: 269--280.

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

John C Nash, and Duncan Murdoch. 2023. *nlsr: Functions for Nonlinear Least Squares Solutions*.

Josef Tvrdík and Ivan Křivý and Ladislav Mišík. 2007. “Adaptive Population-Based Search: Application to Estimation of Nonlinear Regression Parameters.” *Computational Statistics & Data Analysis* 52 (2): 713–24. <https://doi.org/10.1016/j.csda.2006.10.014>.

Levenberg, Kenneth. 1944. “A Method for the Solution of Certain Non-Linear Problems in Least Squares.” *Quarterly of Applied Mathematics* 2: 164--168.

Marquardt, Donald W. 1963. “An Algorithm for Least-Squares Estimation of Nonlinear Parameters.” *SIAM Journal on Applied Mathematics* 11 (2): 431–41.

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

Nash, John C. 1977. “Minimizing a Nonlinear Sum of Squares Function on a Small Computer.” *Journal of the Institute for Mathematics and Its Applications* 19: 231–37.

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

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

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

Nash, John C, and Ravi Varadhan. 2011. *Optimx: A Replacement and Extension of the optim() Function*. Nash Information Services Inc.; Johns Hopkins University.

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

National Institutes for Standards, original from, and Technology http://www.itl.nist.gov/div898/strd/nls/nls\_main.shtml R port by Douglas Bates. 2012. *NISTnls: Nonlinear Least Squares Examples from NIST*. <https://CRAN.R-project.org/package=NISTnls>.

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

Padfield, Daniel, and Granville Matheson. 2020. *Nls.multstart: Robust Non-Linear Regression Using AIC Scores*. <https://CRAN.R-project.org/package=nls.multstart>.

Pinheiro, Jose, Douglas Bates, Saikat DebRoy, Deepayan Sarkar, and R Core Team. 2013. *Nlme: Linear and Nonlinear Mixed Effects Models*.

Prates, Marcos, Victor Lachos, and Aldo Garay. 2021. *Nlsmsn: Fitting Nonlinear Models with Scale Mixture of Skew-Normal Distributions*. <https://CRAN.R-project.org/package=nlsmsn>.

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

Rodriguez-Arias, Mariano, Juan Antonio Fernandez, Javier Cabello, and Rafael Benitez. 2020. *Nlstac: An r Package for Fitting Separable Nonlinear Models*. <https://CRAN.R-project.org/package=nlstac>.

Sokol, Serguei. 2022. *Nlsic: Non Linear Least Squares with Inequality Constraints*. <https://CRAN.R-project.org/package=nlsic>.

Tvrdík, Josef. 2016. *Crsnls: Nonlinear Regression Parameters Estimation by ’CRS4HC’ and ’CRS4HCe’*. <https://CRAN.R-project.org/package=crsnls>.

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

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