An introduction to nlsr

John C. Nash

2023-09-02

R has several tools for estimating nonlinear models and minimizing sums of squares functions. The base R distribution includes a function `nls()` that is feature-rich but has some annoying weaknesses (Nash and Bhattacharjee (2022)). Some of these are addressed in the package `nlsr` (John C Nash and Duncan Murdoch (2019)), for which this vignette provides an introduction with examples.

What does nlsr do?

`nlsr` has functions to find a set of parameters that minimizes the sum of squares (or deviance) of differences between a model specified either as a formula using function `nlxb()` or as both residual and Jacobian R functions using function `nlfb()`. As described below, the Jacobian may be approximated by particular control settings.

Existing R tools that are similar to `nlxb()` are `nls()` and `minpack.lm::nlsLM`; `minpack.lm::nls.lm()` is similar to `nlfb()`.

There are several important differences in approach and results, some of which are detailed in this document.

Output object

The output object of `nlxb()` is smaller than that of `nls()` and is focused on the solution and its properties rather than the modeling task. That is, 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. `nls()` and `nlsLM` return an object of class `nls`, which 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.

Jacobian calculation

Internally, `nlxb()` uses symbolic and automatic differentiation tools to create the residual and Jacobian functions needed for `nlfb()` then uses it to solve the relevant nonlinear least squares minimization problem.

As with other nonlinear least squares packages, we provide the Jacobian code as the “gradient” attribute of the Jacobian function. This lets us embed the code for the Jacobian as this attribute of the residual function so that the call to `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.

Generally `nls()` and `nlsLM()` use a fairly simple forward difference approximation of derivatives for the Jacobian, though a central approximation can be specified in control parameters. Package `nlsr` provides four approximation options, with a further control `ndstep` for the size of the step used in the approximation.

Stabilization of Gauss-Newton computations

The iteration in all the major tools mentioned is the solution of the Gauss-Newton equations. `nls()` uses a variant of an approach suggested by Hartley (1961), while `nlsr` and `minpack.lm` use variants of the method
published by Marquardt (1963), though it was suggested earlier by Levenberg (1944). The details of the method in \texttt{nlsr} are discussed in Nash and Bhattacharjee (2022). Though it is unlikely to be useful in general, control settings for \texttt{nlxb()} or \texttt{nlfb()} allow for those routines to perform a variety of Hartley and Marquardt algorithms. This has been useful for learning about the algorithms, but tangential to simply finding parameters of nonlinear models.

In general, the Levenberg-Marquardt stabilization is important in obtaining solutions. The default method of \texttt{nls()} terminates with \textit{singular gradient} unnecessarily in too many instances.

**Programming language**

An important choice made in developing \texttt{nlsr} was to code entirely within the R programming language. \texttt{nls()} uses a mix of R, C and Fortran, as does \texttt{minpack.lm}. Generally, I believe that keeping to a single programming language allows for easier maintenance and upgrades. On the other hand, R is usually somewhat slower in performing computations because it keeps track of names and structures and because it is usually interpreted rather than compiled. 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, so that using an all-R package offers acceptable performance. Indeed, in \texttt{nlsr} the use of R may be less of a performance cost than the attempt to ensure the solution has been found, which forces more iterations to be used.

**An illustrative example**

The Hobbs weed infestation problem (Nash (1979, 120)) is a growth curve modeling task which is seemingly straightforward but turns out to be quite nasty.

The data and a graph of it is given below.

```r
tt <- 1:12
weedd <- data.frame(tt, weed)
plot(weedd, main="Hobbs weed infestation data")
```

![Hobbs weed infestation data](image)
Most nonlinear least squares problems encountered by R users have data, but it is not strictly required, as we shall see below in the section on functional specification of nonlinear least squares problems. What is required is a definition of the (residual) functions that are to be squared and then their squares summed to provide the loss function or objective function to be minimized by adjusting the parameters that define the set of functions. In the case of the Hobbs problem, I was told the scientists who collected the data believe that the growth of weeds followed a 3-parameter logistic sigmoid growth curve where \( y \) (the weed data) changes with time \( (t) \) (the \( tt \) index) according to the function

**Logistic3U:**

\[
y \approx \frac{b_1}{1 + b_2 \exp(-b_3 \cdot t)}
\]

The problem (using the 3 parameter logistic form above) has very bad scaling and regions in the parameter space where the sum of squares objective is nearly “flat”, so that its Hessian, or matrix of second derivatives, is almost singular. When this occurs, methods cannot easily make progress towards the optimal solution.

Solution methods also need an initial guess for the parameters to be adjusted. There are often some ways to provide such initial parameters, though in my experience they take quite a lot of work to set up reliably, but R does have some worthwhile possibilities in some, though not all, of the selfStart modelling functions, especially those in the package nlraa (Miguez (2021)).

There are, however, alternatives to the model above. For example, a simple scaling can make the solution to the problem easier to find, such as

**Logistic3S:**

\[
y \approx 100 \cdot c_1 / (1 + 10 \cdot c_2 \exp(-0.1 \cdot c_3 \cdot t))
\]

Another form is

**Logistic3T:**

\[
y \approx \text{Asym} / (1 + \exp((xmid - t) / \text{scal}))
\]

The functions above are equivalent. Their parameters are related as follows:

\[
\begin{align*}
\text{Asym} &= b_1 = 100 \cdot c_1 \\
\exp(xmid / \text{scal}) &= b_2 = 10 \cdot c_2 \\
1 / \text{scal} &= b_3
\end{align*}
\]

nlstlr tries to find solutions from very poor initial parameter values. While this is not always possible, nls1r has generally been able to succeed in finding solutions, even when all parameters are started at 1. Let us compare its results with those of nls() and nlsLM().

**Problem setup**

```r
# 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)
```

Solution attempts with nls()
unls1<-try(nls(formula=frmu, start=stu1, data=weeddf))
## Error in nls(formula = frmu, start = stu1, data = weeddf) :
## singular gradient
summary(unls1)
## Length Class Mode
## 1 try-error character
snls1<-try(nls(formula=frms, start=sts1, data=weeddf))
## Error in nls(formula = frms, start = sts1, data = weeddf) :
## singular gradient
summary(snls1)
## Length Class Mode
## 1 try-error character
tnls1<–try(nls(formula=frmt, start=stt1, data=weeddf))
## Error in nls(formula = frmt, start = stt1, data = weeddf) :
## singular gradient
summary(tnls1)
## Length Class Mode
## 1 try-error character

cat("\n")

Solution attempts with nlsr tools

library(nlsr)
unlx1<-try(nlxb(formula=frmu, start=stu1, data=weeddf))
print(unlx1)
## residual sum-squares = 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-09 101
## 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
pshort(unlx1)  # A short form output
## unlx1 -- ss= 2.5873 :  b1 = 196.19  b2 = 49.092  b3 = 0.31357; 25 res/ 19 jac

snlx1<-try(nlxb(formula=frms, start=sts1, data=weeddf))
# pshort(snlx1)
print(snlx1)
## residual sum-squares = 2.5873 on 12 observations
## after 23 Jacobian and 34 function evaluations
## name coeff SE tstat pval gradient JSingval
## c1 1.96186 0.1131 17.35 3.167e-08 7.809e-08 101
## c2 4.90916 0.1688 29.08 3.284e-10 3.578e-07 6.165
## c3 3.1357 0.06863 45.69 5.768e-12 -3.459e-07 2.735
tnlx1<-try(nlxb(formula=frmt, start=stt1, data=weeddf))
# pshort(tnlx1)
print(tnlx1)

## residual sum squares = 2.5873 on 12 observations
## after 27 Jacobian and 36 function evaluations
## name     coeff  SE   tstat   pval    gradient  JSingval
## Asym  196.186 11.31 17.35  3.167e-08  -3.432e-10   44.93
## xmid  12.4173 0.3346 37.11  3.716e-11  -5.273e-08   15.6
## scal  3.18908 0.0698 45.69  5.768e-12   1.752e-07   0.0474

c <- as.list(tnlx1$coefficients)  # Need a list to use ct$... in next line
cat("exp(xmid/scal)=",
exp(c$tmid/c$scal),"
")

## exp(xmid/scal) = 49.092

cat("\\n")

# explicit residuals (weighted if there are weights)
rtnlx1<-residuals(tnlx1)
print(rtnlx1)

## [1] 0.011900 -0.032755  0.092030  0.208782  0.392634 -0.057594 -1.105728
## [8]  0.715786 -0.107648 -0.348396  0.652593  0.287568
## attr("gradient")
##     Asym  xmid  scal
## [1,] 0.027117 -1.6229 5.8103
## [2,] 0.036737 -2.1769 8.5628
## [3,] 0.049596 -2.8997 8.5628
## [4,] 0.066645 -3.8266 10.1000
## [5,] 0.089005 -4.9881 11.6015
## [6,] 0.117921 -6.3988 12.8762
## [7,] 0.154635 -8.0418 13.6607
## [8,]  0.200186 -9.8498 13.6432
## [9,]  0.255106 -11.6901 12.5267
## [10,]  0.319083 -13.3660 10.1313
## [11,]  0.390688 -14.6444  6.5083
## [12,]  0.467334 -15.3139  2.0039

cat("explicit sum squares ="," sum(rtnlx1^2),"\\n")

## explicit sum squares = 2.5873

Solution attempts with minpack.lm

NOTE: The original version of this vignette passed all package checks for Linux, Windows and (Intel-based) Macintosh computers, but failed on Macintosh machines using the M1 chip.

library(minpack.lm)
unlm1<-try(nlsLM(formula=frmu, start=stu1, data=weeddf))
summary(unlm1)  # Use summary() to get display

##
## Formula: weed ~ b1/(1 + b2 * exp(-b3 * tt))
##
## Parameters:
##     Estimate Std. Error t value Pr(>|t|)
## b1 1.96e+02 1.13e+01 17.4 3.2e-08 ***
## b2 4.91e+01 1.69e+00 29.1 3.3e-10 ***
## b3 3.14e-01 6.86e-03 45.7 5.8e-12 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.536 on 9 degrees of freedom
## Number of iterations to convergence: 17
## Achieved convergence tolerance: 1.49e-08

unlm1 # Note the difference. Use this form to get sum of squares

## Nonlinear regression model
## model: weed ~ b1/(1 + b2 * exp(-b3 * tt))
## data: weeddf
## b1  b2  b3
## 196.186 49.092 0.314
## residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 17
## Achieved convergence tolerance: 1.49e-08

# pnls(unlm1) # Short form of output
snlm1<-try(nlsLM(formula=frms, start=sts1, data=weeddf))
# pnls(snlm1)
summary(snlm1)

## Formula: weed ~ 100 * c1/(1 + 10 * c2 * exp(-0.1 * c3 * tt))
##
## Parameters:
## Estimate Std. Error t value Pr(>|t|)
## c1 1.9619 0.1131 17.4 3.2e-08 ***
## c2 4.9092 0.1688 29.1 3.3e-10 ***
## c3 3.1357 0.0686 45.7 5.8e-12 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.536 on 9 degrees of freedom
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08

tnlm1<-try(nlsLM(formula=frmt, start=stt1, data=weeddf))
if (inherits(tnlm1, "try-error")) {
  cat("Failure to compute solution -- likely singular Jacobian\n")
} else {
  pnls(tnlm1) # short form to give sum of squares
  summary(tnlm1)
}

## tnlm1 -- ss= 9205.4 : Asym = 35.532 xmid = 43376 scal = -2935.4; 39 itns
##
## Formula: weed ~ Asym/(1 + exp((xmid - tt)/scal))
##
## Parameters:
## Estimate Std. Error t value Pr(>|t|)
## Asym 3.55e+01  7.91e+03 0.00 1
## xmid 4.34e+04 3.56e+12 0.00 1
## scal -2.94e+03 2.06e+11 0.00 1
##
## Residual standard error: 32 on 9 degrees of freedom
## Number of iterations to convergence: 39
## Achieved convergence tolerance: 1.49e-08

### Solution attempts with wrapnlsr() wrapper

```r
unlw1 <- try(nlsr(formula = frmu, start = stu1, data = weeddf))
print(unlw1)  # using 'unlxl1' gives name of object as 'x' only
```

```r
snlw1 <- try(nlsr(formula = frms, start = sts1, data = weeddf))
# pshort(snlw1)
print(snlw1)
```

```r
tnlw1 <- try(nlsr(formula = fmt, start = stt1, data = weeddf))
print(tnlw1)
```

## Nonlinear regression model
## model: weed ~ b1/(1 + b2 * exp(-b3 * tt))
## data: structure(list(tt = 1:12, weed = c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948, 75.995, 91.972)), class = "data.frame", row.names = c(NA, -12L))
## b1  b2  b3
## 196.186 49.092 0.314
## residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 2.03e-08

## Nonlinear regression model
## model: weed ~ 100 * c1/(1 + 10 * c2 * exp(-0.1 * c3 * tt))
## data: structure(list(tt = 1:12, weed = c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948, 75.995, 91.972)), class = "data.frame", row.names = c(NA, -12L))
## c1  c2  c3
## 1.96 4.91 3.14
## residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 7.03e-08

## Nonlinear regression model
## model: weed ~ Asym/(1 + exp((xmid - tt)/scal))
## data: structure(list(tt = 1:12, weed = c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948, 75.995, 91.972)), class = "data.frame", row.names = c(NA, -12L))
## Asym  xmid  scal
## 196.19 12.42 3.19
## residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 6.35e-09
 Commentary

There are several details that may be important for users.

- **nlsr** is set up to use `print()` to output standard errors and singular values of the Jacobian (for diagnostic purposes). By contrast, **minpack.lm** and **nls()** use `summary()`, which does NOT display the sum of squares, while `print()` gives the sum of squares, but not the standard errors.
- The singular values displayed by `print.nlsr()` (the internal name for the adaptation of the generic `print()` are displayed in a column to the right of the coefficient and standard error display, but are NOT specific to the parameters. Their position is purely for efficient use of page space.
- The most common use of the singular values is to gauge how “nearly singular” the Jacobian is at the solution, and the ratio of the largest to smallest of the singular values is a simple but effective measure. In the above example, we note that the scaled logistic has the smallest ratio.
- The result from **nlsLM** for the transformed model has a very large sum of squares, which may suggest that the program has failed. Since neither **nls()** nor **nlsLM()** offer the singular values, we can “cheat” and use `nlxb()`, though the Jacobian used will be the analytic one used by this last program rather than the forward difference approximation that is generally used by the others.

```r
stspecial <- c(Asym = 35.532, xmid = 43376, scal = -2935.4)
# force to exit at once by setting femax to 1 (maximum number of sum of squares evaluations)
getsvs <- nlxb(formula=frmt, start=stspecial, data=weeddf, control=list(femax=1))
print(getsvs)
```

```
## residual sumsquares = 9205.4 on 12 observations
## after 1 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
```

- The trick used above to get singular values is convenient, but it is actually quite easy to get the Jacobian from a class **nls** object such as **tnlm1** as follows.

```r
if (inherits(tnlm1, "try-error")) {
  cat("Cannot compute solution -- likely singular Jacobian\n")
} else {
  Jtm <- tnlm1$m$gradient()
  svd(Jtm)$d # Singular values
}
```

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 this set of parameters, **nlxb** also stops.

```r
stspecial <- c(Asym = 35.532, xmid = 43376, scal = -2935.4)
# force to exit at once by setting femax to 1 (maximum number of sum of squares evaluations)
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
```
selfStart options

The form of the logistic given by \texttt{frmt} is available as a \texttt{selfStart} model, needing no starting parameters with \texttt{nls()}. The code is in base R in location \texttt{src/library/stats/R/zzModels.R}.

```r
frmtss <- weed ~ SSlogis(tt, Asym, xmid, scal)
ssts1 <- nls(formula=frmtss, data=weeddf)
summary(ssts1)
```

```r
##
## Formula: weed ~ SSlogis(tt, Asym, xmid, scal)
##
## Parameters:
##         Estimate Std. Error t value Pr(>|t|)
## Asym   196.1862    11.3069  17.40 3.2e-08 ***
## xmid   12.4173     0.3346  37.10 3.7e-11 ***
## scal   3.1891      0.0698  45.70 5.8e-12 ***
## ---
## Signif. codes:  
##  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.536 on 9 degrees of freedom
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 1.12e-06
```

```r
require(minpack.lm)
sstm1 <- nlsLM(formula=frmtss, data=weeddf)
summary(sstm1)
```

```r
##
## Formula: weed ~ SSlogis(tt, Asym, xmid, scal)
##
## Parameters:
##         Estimate Std. Error t value Pr(>|t|)
## Asym   196.1863    11.3069  17.40 3.2e-08 ***
## xmid   12.4173     0.3346  37.10 3.7e-11 ***
## scal   3.1891      0.0698  45.70 5.8e-12 ***
## ---
## Signif. codes:  
##  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.536 on 9 degrees of freedom
##
## Number of iterations to convergence: 1
## Achieved convergence tolerance: 1.49e-08
```

The essence of selfStart models is to provide starting parameters for the iterative methods used to find the final parameters. However, an examination of the code for the \texttt{SSlogis()} function shows that it makes use of a DIFFERENT solver and returns the final results of that solver. The computational effort in doing this is, I believe, greater than that expended by \texttt{n1xb()} from an extremely crude start (all 1’s). An alternative selfStart for this version of the logistic is included in package \texttt{nlsr} as \texttt{SSlogisJN.R}

Users should be aware that the programming of selfStart models involves quite esoteric aspects of R which I find prone to errors and difficult to grasp easily. Developers who have built them for us deserve our thanks. A particularly important collection of such tools is in the package Miguez (2021).
SSlogisJN.R for the 3-parameter logistic

SSlogisJN.R uses ideas for selfStart from Ratkowsky (1983) for the Logistic3T form of the model. It produces only an approximate set of starting parameters, unlike SSlogis.R. The core of the idea is as follows, where we use the abbreviated output function to save page space.

```R
Asymt <- 2*max(weed)
Asymt
## [1] 183.94

pw <- Asymt/weed
pw1<-pw-1
lpw1<-log(pw1)
clin <- coef(lm(lpw1~tt))
clin<-as.list(clin)
scalt <- -1/clin$tt
scalt
## [1] 3.1356

xmidt <- scalt*as.numeric(clin[1])
xmidt
## [1] 12.066

library(nlsr)
try1<-nlxb(frmt, data=weeddf, start=c(Asym=1, xmid=1, scal=1))
pshort(try1)
## try1 -- ss= 2.5873 : Asym = 196.19 xmid = 12.417 scal = 3.1891; 36 res/ 27 jac

try2<-nlxb(frmt, data=weeddf, start=c(Asym=Asymt, xmid=xmidt, scal=scalt))
pshort(try2)
## try2 -- ss= 2.5873 : Asym = 196.19 xmid = 12.417 scal = 3.1891; 5 res/ 5 jac
```

We see a dramatic reduction in the computational effort as measured by residual and Jacobian evaluations.

Running selfStart models with nlxb()

There are two important steps in using selfStart models with nlxb():

- we must provide the starting parameters explicitly to nlxb(), which seems counter to the goal of selfStart models. However, we use function getInitial() to exploit particular selfStart modeling functions.
- we must point to a mechanism to compute the Jacobian, since the selfStart function is unlikely to be in the derivatives table of analytic or automatic derivatives. Note that Jacobian (“gradient”) code is generally part of selfStart functions. If code is included for the analytic Jacobian, it is likely worth using. However, because it is easy to make errors in coding derivatives, it may also be worth checking results of such code, for example, using tools from package numDeriv. I have made more such errors over my career than I like to admit.

These steps can be automated, and this has been carried out in function nlsrSS(). The automation is built into the functions nls() and minpack.lm::nlsLM(), but I have preferred to make the process explicit and give it a separate function.

Let us illustrate how to use the getInitial() function (part of base R) to do this.
frmtssJ <- weed ~ SSlogisJN(tt, Asym, xmid, scal) # The model formula
lstrt <- getInitial(frmtssJ, weeddf) # Here we get the starting parameters
cat("starting parameters:
")
## starting parameters:
print(lstrt)
## Asym   xmid   scal
## 183.94 12.066 3.1356
# Next line uses the start. We indicate that the "approximate" Jacobian code is in
# the selfStart code, though in fact it is not an approximation in this case.
sstx1< nlxb(formula=frmtssJ, start=lstrt, data=weeddf, control=list(japprox="SSJac"))
print(sstx1)
## residual sumsquares = 2.5873 on 12 observations
## after 5 Jacobian and 5 function evaluations
## name  coeff    SE  tstat  pval       gradient    JSingval
## Asym  196.186 11.31 17.35 3.167e-08 -4.163e-09   44.93
## xmid  12.4173 0.3346 37.11 3.716e-11 -9.691e-07   15.6
## scal  3.18908 0.0698 45.69 5.768e-12  3.202e-06   0.0474
# If no Jacobian code is included in selfStart module, we can actually use an
# approximate Jacobian. See "gradient" elements for differences in result.
sstx1a< nlxb(formula=frmtssJ, start=lstrt, data=weeddf, control=list(japprox="jacentral"))
print(sstx1a)
## residual sumsquares = 2.5873 on 12 observations
## after 5 Jacobian and 5 function evaluations
## name  coeff    SE  tstat  pval       gradient    JSingval
## Asym  196.186 11.31 17.35 3.167e-08 -4.25e-09   44.93
## xmid  12.4173 0.3346 37.11 3.716e-11 -9.637e-07   15.6
## scal  3.18908 0.0698 45.69 5.768e-12  3.193e-06   0.0474
## We can automate the above in function nlsrSS()
sstSS< nlsrSS(formula=frmtssJ, data=weeddf)

## suggested start=
## Asym   xmid   scal
## 183.94 12.066 3.1356
print(sstSS)

## residual sumsquares = 2.5873 on 12 observations
## after 5 Jacobian and 5 function evaluations
## name  coeff    SE  tstat  pval       gradient    JSingval
## Asym  196.186 11.31 17.35 3.167e-08 -4.163e-09   44.93
## xmid  12.4173 0.3346 37.11 3.716e-11 -9.691e-07   15.6
## scal  3.18908 0.0698 45.69 5.768e-12  3.202e-06   0.0474

Starting parameters for the Logistic3U model
A similar approach for the Logistic3U model can be used.

b1t <- 2*max(weed)
b1t
## [1] 183.94
\[
\begin{align*}
lpw1 & \leftarrow \log(\text{bit/weed-1}) \\
\text{clin} & \leftarrow \text{as.list(coef(lm(lpw1-tt)))} \\
b3t & \leftarrow -\text{clin}$tt \\
b3t & \quad \quad \text{[1] 0.31892} \\
b2t & \leftarrow \exp(\text{as.numeric(clin[1])}) \\
b2t & \quad \quad \text{[1] 46.904} \\
\text{library(nlsr)} \\
\text{try1} & \leftarrow \text{nlxb(frmu, data=weeddf, start=c(b1=1, b2=1, b3=1))} \\
\text{pshort(try1)} & \quad \quad \text{try1 -- ss= 2.5873 : } b1 = 196.19 \quad b2 = 49.092 \quad b3 = 0.31357; 25 \text{ res/ 19 \text{ jac}} \\
\text{try2} & \leftarrow \text{nlxb(frmu, data=weeddf, start=c(b1=b1t, b2=b2t, b3=b3t))} \\
\text{pshort(try2)} & \quad \quad \text{try2 -- ss= 2.5873 : } b1 = 196.19 \quad b2 = 49.092 \quad b3 = 0.31357; 5 \text{ res/ 5 \text{ jac}} \\
\end{align*}
\]

Once again, there is a significant saving in the number of iterations. We have not coded a selfStart function for Logistic3U, preferring to use the scaled form Logistic3S for which we do not need good starting values.

**Jacobian computation**

`nlxb()` offers more options for how the Jacobian is computed than either `nls()` or `minpack.lm::nlsLM()`. The choice is made using the `control` list element `japprox`. If this is NOT specified, `nlxb()` attempts to build Jacobian code using analytic or automatic derivative codes. This is not, of course, always possible, and sometimes we will get an “error” message that required information is not in the derivatives table. This will be the case when we use a function like `SSlogisJN()` unless we indicate that the code is available from a selfStart module by using a value of `SSJac` for `japprox`. The example also showed how we can specify an approximation method, which is also useful when a formula does not have analytic derivatives available. In such cases `control` element `japprox` can be set to one of `jafwd`, `jaback`, `jacentral` or `jand`, giving forward, backward, central and package `numDeriv` approximations. I recommend `jacentral` as a reasonable compromise between accuracy of approximation and amount of computational work.

**Bounds constraints on parameters**

In some cases, we know that parameters cannot be bigger or smaller than some externally known limits. Cash reserves cannot be negative. Animals have a minimum need for water. Airplanes cannot carry more than a known or legislated cargo weight. Such limits can 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.

- `nlsLM()` includes bounds in the standard call, but I have observed cases where it fails to get the correct answer. From my examination of the code, I believe the authors have not taken into account all possibilities, though it should be noted that I regard all programs as having some weakness in regard to constrained optimization. Software creators have to work with assumptions on the extremity of scale that they are willing to countenance, and sometimes problems will be outside the scope envisaged.
# Start MUST be feasible i.e. on or within bounds

```r
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)
```

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

```r
# nlsLM seems NOT to work with bounds in this example

```r
anlsLM1b <- nlsLM(frms, start=sts1, data=weeddf, lower=c(0,0,0), upper=c(2,6,3))
```

```
pnls(anlsLM1b)
```

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

```r
# also no warning if starting out of bounds, but gets good answer!!

```r
st4<-c(c1=4, c2=4, c3=4)
```

```r
anlsLMob <- nlsLM(frms, start=st4, data=weeddf, lower=c(0,0,0), upper=c(2,6,3))
```

```
pnls(anlsLMob)
```

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

```r
# Try nlsr::nlxb()

```r
anlx1b <- nlxb(frms, start=sts1, data=weeddf, lower=c(0,0,0), upper=c(2,6,3))
```

```
pshort(anlx1b)
```

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

## Functional specification of nonlinear least squares problems

We illustrate how to solve nonlinear least squares problems where the set of functions to be squared is provided by an R function, as is the Jacobian matrix. Note that nlsr::nlfb() gets this information from the object returned by the jacfn argument in the “gradient” attribute of that object. This is a programming artifice to allow a simplification of the nlxb() code. The example is the well-known Banana shaped valley of Rosenbrock (1960).

```r
frres<-function(x) { ## Rosenbrock as residuals
  x1 <- x[1]
  x2 <- x[2]
  res<-c( 10 * (x2 - x1 * x1), (1 - x1) )
}
```

```r
frjac<-function(x) { ## Rosenbrock residual derivatives matrix
  x1 <- x[1]
  x2 <- x[2]
  J<-matrix(NA, nrow=2, ncol=2)
  J[1,1]<- -20*x1
  J[1,2]<- 10
  J[2,1]<- -1
  J[2,2]<- 0
  attr(J,"gradient")<-J # NEEDED for nlfb()
  J
}
```

```r
require(minpack.lm)
require(nlsr)
strt<-c(-1.2,1)
rosnlf<-nlfb(strt, resfn=frres, jacfn=frjac)
print(rosnlf)
```

```r
# Start MUST be feasible i.e. on or within bounds
```

```r
# Functional specification of nonlinear least squares problems
```
## residual sumsquares = 5.6364e-16 on 2 observations
## after 19 Jacobian and 27 function evaluations

<table>
<thead>
<tr>
<th>name</th>
<th>coeff</th>
<th>SE</th>
<th>tstat</th>
<th>pval</th>
<th>gradient</th>
<th>JSingval</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>1</td>
<td>Inf</td>
<td>0</td>
<td>NaN</td>
<td>-3.212e-09</td>
<td>22.38</td>
</tr>
<tr>
<td>p2</td>
<td>1</td>
<td>Inf</td>
<td>0</td>
<td>NaN</td>
<td>-1.025e-08</td>
<td>0.4469</td>
</tr>
</tbody>
</table>

rosnlm <- nls.lm(par=strt, fn=frres, jac=frjac)

summary(rosnlm)

## Parameters:
## Estimate Std. Error t value Pr(>|t|)
## [1,] 1 NaN NaN NaN
## [2,] 1 NaN NaN NaN

## Residual standard error: NaN on 0 degrees of freedom
## Number of iterations to termination: 16
## Reason for termination: The cosine of the angle between \`fvec\' and any column of the Jacobian is at most \`gtol\' in absolute value.

I find it awkward to solve problems specified this way with nls() and do not believe it is worth pursuing that topic.

### Weighted nonlinear regression

Regression attempts to explain a dependent (or predicted) variable as a function of independent (or explanatory or predictor) variables and estimated parameters. The estimation method commonly used is minimization of the sum of squared residuals. However, these residuals may be weighted. In the tools available in R for nonlinear least squares, weights can be specified which multiply the squared residuals, and we need a weight for each term in the sum. Each residual is therefore multiplied by the square root of its respective weight.

Typically, the weights are given as a vector of numbers. A popular choice is the reciprocal of a measure of the variance of each data observation. Replicated data allows for sample variances to be used; otherwise we need some proxy. The concept is that the weights are proportional to our belief that the observations are correct.

#### Static weights

Let us demonstrate with weights that are simply the reciprocal of the independent variable. This may not make sense for statistical modeling, but it lets us see that the residuals() function returns weighted residuals.

wts <- 1/weeddf$tt # wts are reciprocal of time value
tnlx1w <- try(nlxb(formula=frmt, start=stt, data=weeddf, weights=wts))  # pshort(tnlx1w)

print(tnlx1w)

## residual sumsquares = 0.34107 on 12 observations
## after 24 Jacobian and 31 function evaluations

<table>
<thead>
<tr>
<th>name</th>
<th>coef</th>
<th>SE</th>
<th>tstat</th>
<th>pval</th>
<th>gradient</th>
<th>JSingval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asym</td>
<td>194.292</td>
<td>9.858</td>
<td>19.71</td>
<td>1.033e-08</td>
<td>-4.585e-11</td>
<td>18.07</td>
</tr>
<tr>
<td>xmid</td>
<td>12.3603</td>
<td>0.2849</td>
<td>43.39</td>
<td>9.163e-12</td>
<td>-4.422e-09</td>
<td>5.737</td>
</tr>
<tr>
<td>scal</td>
<td>3.17704</td>
<td>0.05032</td>
<td>63.13</td>
<td>3.166e-13</td>
<td>1.007e-08</td>
<td>0.01974</td>
</tr>
</tbody>
</table>

c<--as.list(tnlx1w$coefficients)  # Need a list to use ct$... in next line
cat("exp(xmid/scal)=",exp(ct$xmid/ct$scal),"\n")

## exp(xmid/scal) = 48.936
rtnlx1w <- residuals(tnlx1w)
print(rtnlx1w)

## [1] -0.017057 -0.045307  0.034645  0.089356  0.164682 -0.029318 -0.417824
## [8]  0.259068 -0.025608 -0.099651  0.200705 -0.094874
## attr(,"gradient")
## Asym  xmid   scal
## [1,]  0.027232 -1.6200  5.7928
## [2,]  0.036934 -2.1753  7.0935
## [3,]  0.049915 -2.9002  8.5446
## [4,]  0.067140 -3.8303 10.0793
## [5,]  0.089747 -4.9959 11.5742
## [6,]  0.118997 -6.4113 12.8352
## [7,]  0.156144 -8.0580 13.5955
## [8,]  0.202225 -9.8662 13.5408
## [9,]  0.257752 -11.7000 12.3749
## [10,]  0.322364 -13.3591  9.9248
## [11,]  0.394564 -14.6090  6.2551
## [12,]  0.471678 -15.2397  1.7283

cat("explicit sumsquares =", sum(rtnlx1w^2),"\n")

## explicit sumsquares = 0.34107

Weights that are functions of the model parameters

Possible choices for weights include the reciprocal of the squared residuals or squared fitted values. Unfortunately, such quantities depend on the parameter values. Consider our Logistic example. We want to predict \( y \) with

\[
\text{fitted}(b_1, b_2, b_3, t) = \frac{b_1}{1 + b_2 \times \exp(-b_3 \times t)}
\]

Thus the raw residuals are

\[
\text{rawres}(b_1, b_2, b_3, t) = y - \text{fitted}(b_1, b_2, b_3, t)
\]

But if the weights are \((1/\text{fitted}(b_1, b_2, b_3, t))\), then the residuals for which the sum of squares is to be calculated are

\[
\text{resid}(b_1, b_2, b_3, t) = \text{rawres}(b_1, b_2, b_3, t) \times (1/\text{fitted}(b_1, b_2, b_3, t))
\]

\[
= \frac{y}{(b_1/(1 + b_2 \times \exp(-b_3 \times t)))} - 1
\]

However, we still need to keep in mind that the user needs to use the formula for the fitted values to get predictions. While the software needs to minimize the weighted (hence modified) sum of squares, it also must keep track of the fitted/predicted values.

\texttt{minpack.lm} offers a function \texttt{wfct()} that can be used within the argument supplied as \texttt{weights} in the call to \texttt{nlsLM()} or \texttt{nls()}, though it usually fails with \texttt{nlsr::nlxb()}. The action of this function was surprising to me, as \texttt{wfct()} takes an expression as its argument (and may itself be part of an expression in the argument after the = sign). If the argument to \texttt{wfct} contains \texttt{fitted}, \texttt{resid} or \texttt{error}, then the call to \texttt{nlsLM} or \texttt{nls} will trigger a further call to one of these functions, evaluate the \texttt{weights} and then run again to compute the
solution. This can be seen by setting \( \text{trace} = \text{TRUE} \). The process can be verified explicitly for the example used in the \texttt{wfct} documentation. Unfortunately, \texttt{wfct()} appears to crash \texttt{knitr} or \texttt{rmarkdown}, so we have evaluated the example outside of Rstudio and included the output below.

```r
library(minpack.lm)
lapply(nlsr) \# for \texttt{pnls}
Treated <- Puromycin[Puromycin$state == "treated", ]
# First get raw estimates
wtt3nlm0 <- nlsLM(rate ~ Vm * conc/(K + conc), data = Treated, trace=TRUE,
                  start = c(Vm = 200, K = 0.05))
fit0 <- fitted(wtt3nlm0)
# Static wts using \textit{fit0}
wtt3nlm0s <- nlsLM(rate ~ Vm * conc/(K + conc), data = Treated, trace=TRUE,
                  start = c(Vm = 200, K = 0.05), weights = wfct(1/fitted^2))
pnls(wtt3nlm0s)
# and run directly, noting the 2 phase operation
wtt3nlm <- nlsLM(rate ~ Vm * conc/(K + conc), data = Treated, trace=TRUE,
                 start = c(Vm = 200, K = 0.05), weights = wfct(1/fitted^2))
pnls(wtt3nlm)
cat("weights from wtt3nlm\n")
as.numeric(wtt3nlm$weights)
```

```
It.  0, RSS = 1636.59, Par. = 200 0.05
It.  1, RSS = 1205.62, Par. = 211.157 0.0616271
It.  2, RSS = 1195.57, Par. = 212.511 0.0638418
It.  3, RSS = 1195.45, Par. = 212.666 0.0640939
It.  4, RSS = 1195.45, Par. = 212.682 0.0641186
It.  5, RSS = 1195.45, Par. = 212.684 0.064121

It.  0, RSS = 0.22811, Par. = 200 0.05
It.  1, RSS = 0.227222, Par. = 200.913 0.0494747
It.  2, RSS = 0.227221, Par. = 200.847 0.0494303
It.  3, RSS = 0.227221, Par. = 200.842 0.049426
It.  4, RSS = 0.227221, Par. = 200.841 0.0494256
wtt3nlm0s -- ss= 0.2272213 : Vm = 200.841 K = 0.04942558; 4 itns
It.  0, RSS = 1636.59, Par. = 200 0.05
It.  1, RSS = 1205.62, Par. = 211.157 0.0616271
It.  2, RSS = 1195.57, Par. = 212.511 0.0638418
It.  3, RSS = 1195.45, Par. = 212.666 0.0640939
It.  4, RSS = 1195.45, Par. = 212.682 0.0641186
It.  5, RSS = 1195.45, Par. = 212.684 0.064121
It.  0, RSS = 0.22811, Par. = 200 0.05
It.  1, RSS = 0.227222, Par. = 200.913 0.0494747
It.  2, RSS = 0.227221, Par. = 200.847 0.0494303
It.  3, RSS = 0.227221, Par. = 200.842 0.049426
It.  4, RSS = 0.227221, Par. = 200.841 0.0494256
wtt3nlm -- ss= 0.2272213 : Vm = 200.841 K = 0.04942558; 4 itns
Show in New Window
weights from wtt3nlm
```

The documentation of \texttt{wfct} suggests that it can also use the name of the response (dependent) variable or the name of the predictor (independent) variable. However, some models have more than one independent variable, and I have not explored what happens in such cases.
It would be helpful if \textit{nlsr} had the capability to include functional weights. Duncan Murdoch suggested a patch that allows this. We modify the \texttt{nlfb()} routine to detect and act on functional weights.

```r
library(nlsr)
Treated <- Puromycin[Puromycin$state == "treated",]

# "regression" formula
w1frm <- rate ~ Vm * conc/(K + conc)

# Explicit dynamic weighted nlxb
w3frm <- rate/(Vm * conc/(K + conc)) ~ 1
dynweighted <- nlxb(w3frm, data = Treated, start = c(Vm = 201.003, K = 0.04696),
                   trace=TRUE, control=list(prtlvl=0))

## No backtrack
## 00lambda: 1e-04 SS= 0.17941 ( NA ) at Vm = 201 K = 0.04696 f/j 1 / 0
## <<lambda: 4e-05 SS= 0.17941 ( 3.3805e-05 ) at Vm = 201 K = 0.046962 f/j 2 / 1
## <<lambda: 1.6e-05 SS= 0.17941 ( 5.2155e-07 ) at Vm = 201 K = 0.046962 f/j 3 / 2

pshort(dynweighted)

## dynweighted -- ss= 0.17941 : Vm = 201 K = 0.046962; 3 res/ 3 jac

# Compute the model from the ORIGINAL model (w1frm) using parameters from dynweighted
dyn0 <- nlxb(w1frm, start=coefficients(dynweighted), data=Treated, control=list(jemax=0))
wfdyn0 <- 1/fitted(dyn0)^2 # weights

print(as.numeric(wfdyn0))

## [1] 2.7745e-04 2.7745e-04 7.8659e-05 7.8659e-05 5.0396e-05 5.0396e-05

pshort(dyn0)

## dyn0 -- ss= 1786.1 : Vm = 201 K = 0.046962; 1 res/ 1 jac

formulaweighted <- nlxb(w1frm, data = Treated, start = c(Vm = 201.003, K = 0.04696),
                        weights = ~ 1/fitted(formulaweighted)^2, trace=TRUE, control=list(prtlvl=0))

## No backtrack
## 00lambda: 1e-04 SS= 0.17941 ( NA ) at Vm = 201 K = 0.04696 f/j 1 / 0
## <<lambda: 4e-05 SS= 0.17286 ( 0.059872 ) at Vm = 201.77 K = 0.050223 f/j 2 / 1

pshort(formulaweighted)

## formulaweighted -- ss= 0.17286 : Vm = 201.77 K = 0.050223; 19 res/ 2 jac

wtsfromfits<-1/fitted(formulaweighted)^2

print(as.numeric(wtsfromfits))


print(as.numeric(formulaweighted$weights))


There are differences in the (weighted) sums of squares. The \texttt{dynweighted} case minimizes the residuals weighted by the actual CURRENT fits as expressed by the formula. The \texttt{formulaweighted} case computes the weights at each iteration from the previous (or original) fit. It is a form of iteratively weighted least squares. The case \texttt{wtt3nlm} above uses \texttt{nlsLM} with weights specified in \texttt{wfct()} and computes the weights.
once from an unweighted model, then runs a second, statically weighted calculation. Keeping track of these differences requires a lot of care, and I advise documenting what is done carefully.

### Ongoing efforts

The examples above show how to use package `nlsr` as it exists at the end of December 2022. The story, however, is not finished. It would be very nice to be able to identify and work with models that are partially linear. Solving such models is possible with the `algorithm="plinear"` option of `nls()`, but identifying which parameters enter linearly is not easy for most users. Moreover, the specification of the model to the `VARPRO` solver that is called by the `plinear` option is inconsistent with the general specification used by `nls()` and other nonlinear least squares tools for R.

There are also a number of possible tweaks to details in `nlsr` and efforts to include such improvements are a continuing interest. I welcome communication and collaboration to continue the improvement of the package and R in general.

### References


