An introduction to nlsr

John C. Nash

2024-02-11

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 (John C. 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.

There is more discussion of special considerations for specifying the residuals and Jacobian later in the vignette.

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). Though it is unlikely to be useful in general, control settings for `nlxb()` or `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.

Nonlinear least squares methods are mostly founded on some or other variant of the Gauss-Newton algorithm. The function we wish to minimize is the sum of squares of the (nonlinear) residuals $r(x)$ where there are m observations (elements of r) and n parameters x. Hence the function is

$$f(x) = \sum_i ({r_i}^2)$$

Newton's method starts with an original set of parameters $x[0]$. At a given iteraion, which could be the first, we want to solve

$$x[k+1] = x[k] - H^{-1}g$$

where H is the Hessian and g is the gradient at $x[k]$. We can rewrite this as a solution, at each iteration, of

$$H\delta = -g$$

with

$$x[k+1] = x[k] + \delta$$

For the particular sum of squares, the gradient is

$$g(x) = 2 * r[k]$$

and

$$H(x) = 2(J'J + \sum_i (r_i * Z_i))$$

where J is the Jacobian (first derivatives of r w.r.t. x) and Z_i is the tensor of second derivatives of r_i w.r.t. x). Note that J' is the transpose of J.

The primary simplification of the Gauss-Newton method is to assume that the second term above is negligible. As there is a common factor of 2 on each side of the Newton iteration after the simplification of the Hessian, the Gauss-Newton iteration equation is

$$(J'J)\delta = -J'r$$

This iteration frequently fails. The approximation of the Hessian by the Jacobian inner-product is one reason, but there is also the possibility that the sum of squares function is not "quadratic" enough that the unit step reduces it. Hartley (1961) introduced a line search along delta. Marquardt (1963) suggested replacing $J'J$ with $(J'J + \lambda * D)$ where D is a diagonal matrix intended to partially approximate the omitted portion of the Hessian. Choices suggested by Marquardt were $D = I$ (a unit matrix) or $D =$ (diagonal part of $J'J$). The former approach, when λ is large enough that the iteration is essentially

$$\delta = -g/\lambda$$

gives a version of the steepest descents algorithm. Using the diagonal of $J'J$, we have a scaled version of this (see https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm; Levenberg (1944) predated Marquardt, but the latter seems to have done the practical work that brought the approach to general attention.)

Both `nlsr::nlxb()` and `minpack.lm::nlsLM` use a Levenberg-Marquardt stabilization of the iteration described above, with `nlsr` using the modification involving the ϕ control parameter. The complexities of the code in `minpack.lm` are such that I have relied largely on the documentation to judge how the iteration is accomplished. `nls()` uses a straightforward Hartley variant of the Gauss-Newton iteration, but rather than form the sum of squares and cross-products, uses a QR decomposition of the matrix J that has been found by a forward difference approximation. The line search used by `nls()` is a simple back-tracking search using a step reduction factor of 0.5 as the default stepsize reduction.

J. C. Nash (1979) and John C. Nash and Walker-Smith (1987) solve

$$(J^T J + \lambda D_x)\delta = -J^T r$$

by the Cholesky decomposition. In this $D_x = (D + \phi * I)$ is used as above and λ is a number of modest size initially. ϕ is typically 1. Clearly for $\lambda = 0$ we have a Gauss-Newton method. Typically, the sum of squares of the residuals calculated at the "new" set of parameters is used as a criterion for keeping those parameter values. If so, the size of λ is reduced. If not, we increase the size of λ and compute a new δ. Note that a new J, the expensive step in each iteration, is NOT required in this latter case.

In 2022, a modification to use $D_y = (\psi * D + \phi * I)$ was introduced, though the matrix equations are solved via a QR decomposition approach. Within the code, control parameters `psi`, `phi` and `stepredn` were introduced so that a variety of Gauss-Newton, Hartley, or Marquardt approaches are available by simple control modifications. Experience so far suggests that a Levenberg-Marquardt stabilization is much more reliable than the Gauss-Newton-Hartley choices, but that different selections of `psi` and `phi` perform rather similarly. As for Gauss-Newton methods, the details of how to start, adjust and terminate the iteration lead to many variants, increased by these different possibilities for specifying D. See J. C. Nash (1979).

Programming language

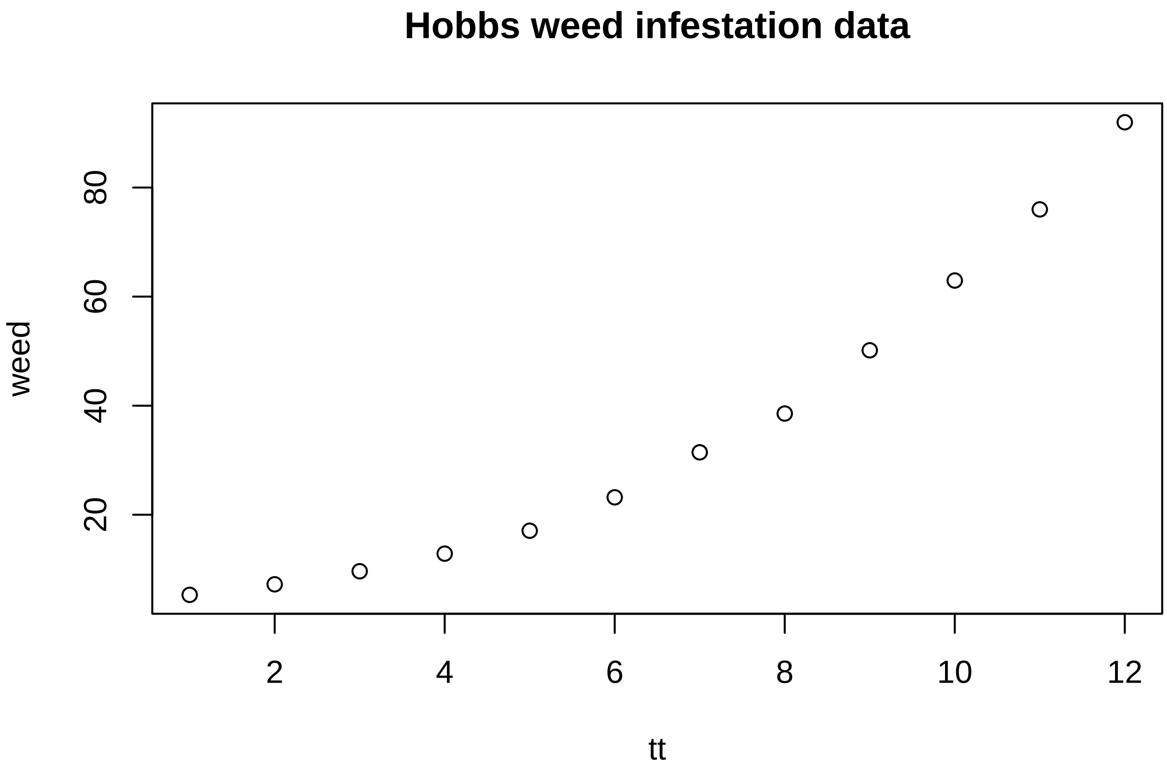
An important choice made in developing `nlsr` was to code entirely within the R programming language. `nls()` uses a mix of R, C and Fortran, as does `minpack.lm`. Generally, I believe that keeping to a single programming language allows for easier maintenance and upgrades. On the other hand, R is usually 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 `nlsr` the use of R may be less of a performance cost than the choice to be aggressive in ensuring a solution has been found, which forces more iterations to be used.

An illustrative example

The Hobbs weed infestation problem (John C. 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.

```
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")
```



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 believed 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:

$$y \approx b_1/(1 + b_2 * exp(-b_3 * t))$$

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

Logistic3S:

$$y \approx 100 * c_1/(1 + 10 * c_2 * exp(-0.1 * c_3 * t))$$

Another form is

Logistic3T:

$$y \approx Asym/(1 + exp((xmid - t)/scal))$$

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

$$Asym = b_1 = 100 * c_1$$
$$exp(xmid/scal) = b_2 = 10 * c_2$$
$$1/scal = b_3$$

`nlsr` is programmed to try to find solutions from very poor initial parameter values. While this is not always possible, `nslr` 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

```
# 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")
```

The “singular gradient” results here, with this problem, were the main motivation for developing `nlsr`. It is actually the Jacobian matrix which is singular, but because the Jacobian is in a sense the “gradient” of the residuals, R has emitted its particular error report.

Solution attempts with nlsr tools

```
library(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
```

```
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 sumsquares =  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       130.1
## 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 sumsquares =  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
```

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

```
## 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  7.1111
##  [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 sumsquares =", sum(rtnlx1^2),"\n")
```

```
## explicit sumsquares = 2.5873
```

nlsr::nlxb() has succeeded in finding a solution in all cases from the poor "all-1s" starts.

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        1
## xmid  4.34e+04   3.56e+12       0        1
## scal -2.94e+03   2.06e+11       0        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

```
## Try the wrapper. Calling wrapnlsr() instead of nlsr() is equivalent
##
unlw1<-try(nlsr(formula=frmu, start=stu1, data=weeddf))
print(unlw1) # using 'unlx1' gives name of object as 'x' only
```

```
## 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, 38.5
##      b1      b2      b3
## 196.186  49.092   0.314
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 5.67e-08
```

```
snlw1<-try(nlsr(formula=frms, start=sts1, data=weeddf))
# pshort(snlx1)
print(snlw1)
```

```
## 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, 38.5
##   c1   c2   c3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 6.25e-08
```

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

```
## 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, 38.5
##   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.

```
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.

```
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.

```
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
```

Extracting standard errors from nlsr solutions

While it is straightforward to display coefficients of models with their standard errors (SE) and related statistics by **print**ing the solution, some users may need to be able to save and use the quantities, for example, in particular presentations of them. A query from Rohana Ambagaspitiya of the University of Calgary prompted me to explain how to do this, as the SE's and related quantities are not in the output of `nlxb()` or `nlfb()`, but are computed in the `summary()` and `print()` functions. Saving the SE of the `Asym` coefficient in the `tnlx1` solution above is illustrated.

```
ssum<-summary(tnlx1)
# str(ssum) # This will display the contents of the summary if we wish.
# The key element is the param array
print(ssum$param)
```

```
##      Estimate Std. Error t value   Pr(>|t|)
## Asym 196.1863  11.306939  17.351 3.1667e-08
## xmid  12.4173   0.334621  37.109 3.7156e-11
## scal   3.1891   0.069801  45.688 5.7676e-12
```

```
AsymSE<-ssum$param[1,2]
AsymSE
```

```
## [1] 11.307
```

selfStart options

The form of the logistic given by `frmt` is available as a **selfStart** model, needing no starting parameters with `nls()` or `minpack.lm::nlsLM()`. The code is in base R in location `src/library/stats/R/zzModels.R`.

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

```
##
## Formula: weed ~ SSlogis(tt, Asym, xmid, scal)
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## Asym 196.1862    11.3069    17.4  3.2e-08 ***
## xmid  12.4173     0.3346    37.1  3.7e-11 ***
## scal   3.1891     0.0698    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: 0
## Achieved convergence tolerance: 1.12e-06
```

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

```
##
## Formula: weed ~ SSlogis(tt, Asym, xmid, scal)
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## Asym 196.1863    11.3069    17.4  3.2e-08 ***
## xmid  12.4173     0.3346    37.1  3.7e-11 ***
## scal   3.1891     0.0698    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: 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 `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 `nlxb()` from an extremely crude start (all 1's). An alternative selfStart for this version of the logistic is included in package `nlsr` as `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) # use double the largest weed value as a guess to Asymt
Asymt
```

```
## [1] 183.94
```

```
pw <- Asymt/weed # intermediate quantities to compute scale and xmidt
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:\n")
```

```
## starting parameters:
```

```
print(lstrt)
```

```
##      Asym     xmid     scal
## 183.9440  12.0660   3.1356
```

```r
# 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
```

```r
# 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
```

```r
## We can automate the above in function nlsrSS()
sstSS<-nlsrSS(formula=frmtssJ, data=weeddf)
```

```
## suggested start=
##     Asym     xmid     scal
## 183.9440  12.0660   3.1356
```

```r
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.

```r
b1t <- 2*max(weed)
b1t
```

```
## [1] 183.94
```

```r
lpw1<-log(b1t/weed-1)
clin <- as.list(coef(lm(lpw1~tt)))
b3t <- -clin$tt
b3t
```

```
## [1] 0.31892
```

```r
b2t <- exp(as.numeric(clin[1]))
b2t
```

```
## [1] 46.904
```

```
library(nlsr)
try1<-nlxb(frmu, data=weeddf, start=c(b1=1, b2=1, b3=1))
pshort(try1)
```

```
## try1  -- ss= 2.5873 :  b1 = 196.19  b2 = 49.092  b3 = 0.31357; 25 res/ 19 jac
```

```
try2<-nlxb(frmu, data=weeddf, start=c(b1=b1t, b2=b2t, b3=b3t))
pshort(try2)
```

```
## try2  -- ss= 2.5873 :  b1 = 196.19  b2 = 49.092  b3 = 0.31357; 5 res/ 5 jac
```

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
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
```

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).

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

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

```
## residual sumsquares =  5.6364e-16  on  2 observations
##     after  19    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                    1          Inf          0        NaN  -3.212e-09       22.38
## p2                    1          Inf          0        NaN  -1.025e-08      0.4469
```

```
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 m
```

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=stt1, data=weeddf, weights=wts))
# pshort(tnlx1)
print(tnlx1w)
```

```
## residual sumsquares =  0.34107  on  12 observations
##     after  24    Jacobian and  31 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Asym             194.292        9.858       19.71  1.033e-08  -4.585e-11       18.07
## xmid             12.3603       0.2849       43.39  9.163e-12  -4.422e-09       5.737
## scal             3.17704      0.05032       63.13  3.166e-13   1.007e-08     0.01974
```

```
ct<-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
```

```
cat("\n")
```

```
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

$$fitted(b_1, b_2, b_3, t) = b_1/(1 + b_2 * exp(-b_3 * t))$$

Thus the raw residuals are

$$rawres(b_1, b_2, b_3, t) = y - fitted(b_1, b_2, b_3, t)$$

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

$$resid(b_1, b_2, b_3, t) = rawres(b_1, b_2, b_3, t) * (1/fitted(b_1, b_2, b_3, t)$$

$$= y/(b_1/(1 + b_2 * exp(-b_3 * t))) - 1$$

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.

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

```
library(minpack.lm)
library(nlsr) # for 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 fit0
wtt3nlm0s<-nlsLM(rate ~ Vm * conc/(K + conc), data = Treated, trace=TRUE,
```

```
                start = c(Vm = 200, K = 0.05), weights = wfct(1/fit0^2))
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
 [1] 3.910941e-04 3.910941e-04 9.460635e-05 9.460635e-05 5.539227e-05 5.539227e-05
 [7] 3.687173e-05 3.687173e-05 2.745956e-05 2.745956e-05 2.475956e-05 2.475956e-05
```

The documentation of `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 `nlsr` had the capability to include functional weights. Duncan Murdoch suggested a patch that allows this. We modify the `nlfb()` routine to detect and act on functional weights.

```
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
## 00lamda: 1e-04  SS= 0.17941 ( NA ) at  Vm = 201  K = 0.04696 f/j  1 / 0
## <<lamda: 4e-05  SS= 0.17941 ( 3.3805e-05 ) at  Vm = 201  K = 0.046962 f/j  2 / 1
## <<lamda: 1.6e-05  SS= 0.17941 ( 5.2155e-07 ) at  Vm = 201  K = 0.046962 f/j  3 / 2
```

```r
pshort(dynweighted)
```

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

```r
# 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
##  [7] 3.6446e-05 3.6446e-05 2.9076e-05 2.9076e-05 2.6910e-05 2.6910e-05
```

```r
pshort(dyn0) # Shows sumsquares without weights, but computes weights we used
```

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

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

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

```r
pshort(formulaweighted)
```

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

```r
wtsfromfits<-1/fitted(formulaweighted)^2
print(as.numeric(wtsfromfits))
```

```
##  [1] 3.0281e-04 3.0281e-04 8.2893e-05 8.2893e-05 5.2112e-05 5.2112e-05
##  [7] 3.7057e-05 3.7057e-05 2.9166e-05 2.9166e-05 2.6857e-05 2.6857e-05
```

```r
print(as.numeric(formulaweighted$weights))
```

```
##  [1] 3.0281e-04 3.0281e-04 8.2893e-05 8.2893e-05 5.2112e-05 5.2112e-05
##  [7] 3.7057e-05 3.7057e-05 2.9166e-05 2.9166e-05 2.6857e-05 2.6857e-05
```

There are differences in the (weighted) sums of squares. The **dynweighted** case minimizes the residuals weighted by the actual CURRENT fits as expressed by the formula. The **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 **wtt3nlm** above uses `nlsLM` with weights specified in `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.

Models that use multiple functional forms

The query by John Sorkin in 2013 on the R-help list, item https://stat.ethz.ch/pipermail/r-help/2013-January/344918.html , asks about fitting a composite model made up of "pieces". Sorkin's problem, as described, seems incomplete. We will use two problems that we can present clearly.

Two straight lines

We will generate data for two intersecting straight lines. These will intersect at the point `(kx, ky)`, with slopes `sl` and `sr` on the left and right hand sides of the point.

```r
xx<-1:50 # simple sequence of 50 points
kx<-24 # Set x coordinate of intersect
ky<-10 # Set y coordinate of intersect
sl<-1 # left slope
sr<-0 # right slope
# now write down model using conditionals
formu<- yy ~ (xx <= kx)*(sl*(xx - kx)+ky) +  (xx > kx)*(sr*(xx - kx)+ky)
# Generate "exact" model for given parameters
yy0<- (xx <= kx)*(sl*(xx - kx)+ky) +  (xx > kx)*(sr*(xx - kx)+ky)
ee<-runif(50, -3, 3) # generate some "errors"
ee<-ee-mean(ee) # but center them
yy<-yy0+ee # and add error to the "exact" model
sldf<-data.frame(xx=xx, yy=yy) # make a data frame of the data
# Try to estimate a model from the data and the formula
t2sl<-try(nlxb(formu, data=sldf, start=c(kx=1, ky=1, sl=1, sr=1), trace=TRUE))
```

```
## Error in deriv.default(residexpr, names(pvec)) :
##   Function '`>`' is not in the derivatives table
```

```r
# But that fails because the ">" in the formula trips up derivative program
# We can still use approximate Jacobian
t2slx<-nlxb(formu, data=sldf, start=c(kx=1, ky=1, sl=1, sr=1), trace=TRUE,
            control=list(japprox="jacentral"))
```

```
## Using approximation  jacentral
##       is.character(ctrl$japprox)= TRUE
## No backtrack
## 00lamda: 1e-04  SS= 26083 ( NA ) at  kx = 1  ky = 1  sl = 1  sr = 1 f/j  1 / 0
## <<lamda: 4e-05  SS= 939.22 ( 0.9203 ) at  kx = 4.7252  ky = -2.7252  sl = 1  sr = 0.44614 f/j  2 / 1
## <<lamda: 1.6e-05  SS= 425.72 ( 0.52047 ) at  kx = 15.95  ky = 2.2568  sl = 1.0466  sr = 0.37774 f/j
## <<lamda: 6.4e-06  SS= 211.12 ( 0.71582 ) at  kx = 22.62  ky = 9.2281  sl = 1.0251  sr = 0.13026 f/j
## <<lamda: 2.56e-06  SS= 139.65 ( 0.41247 ) at  kx = 24.18  ky = 10.234  sl = 0.99249  sr = -0.010282 f
## <<lamda: 1.024e-06  SS= 138.45 ( 0.065967 ) at  kx = 24.415  ky = 10.283  sl = 0.98364  sr = -0.02559
## <<lamda: 4.096e-07  SS= 138.45 ( 0.0015716 ) at  kx = 24.413  ky = 10.279  sl = 0.98364  sr = -0.0255
```

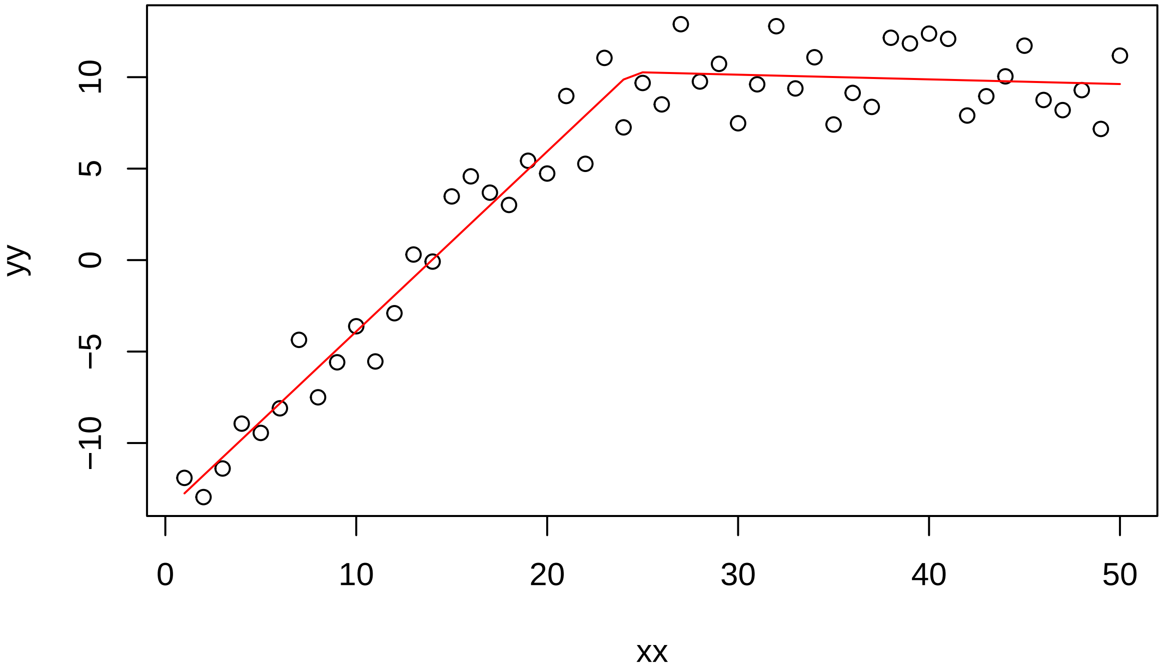
```r
t2slx
```

```
## residual sumsquares =  138.45  on  50 observations
##     after  7    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## kx               24.4133        0.9734      25.08  1.786e-28   2.952e-09       77.04
## ky                10.279        0.6672      15.41  8.788e-20   3.149e-09       67.76
## sl              0.983645       0.05116      19.23  1.272e-23   1.852e-09       3.962
## sr            -0.0255915       0.04537    -0.5641     0.5754   3.217e-07       1.581
```

```r
coef(t2slx)
```

```
##         kx         ky         sl         sr
## 24.413275 10.279001  0.983645 -0.025591
## attr(,"pkgname")
## [1] "nlsr"
```

```r
plot(xx, yy)
yline<-fitted(t2slx)
lines(xx, yline, type='l', col='red')
```



The WOOD test function

Let us try an example – the Wood test function (Problem 14 in Moré, Garbow, and Hillstrom (1981)). Some authors refer to this as the Colville function (https://www.sfu.ca/~ssurjano/colville.html)

This problem is usually stated as a function minimization in 4 parameters. This is

$$f(x) = 100*({x_1}^2 - x_2)^2 + (x_1 - 1)^2 + (x_3 - 1)^2 + 90*({x_3}^2 - x_4)^2 + 10.1((x_2 - 1)^2 + (x_4 - 1)^2) + 19.8*(x_2 - 1)*(x_4 - 1)$$

It is, however, a nonlinear least squares problem for which we can write the residuals as a set of formulas.

```
library(nlsr)
library(minpack.lm)
frm<- 0 ~ (x == 1) * (10 * (`par[2]` - `par[1]` * `par[1]`)) +
  (x == 2) * (1 - `par[1]`) +
  (x == 3) * ((`par[4]` - `par[3]` * `par[3]`) * sqrt(90)) +
  (x == 4) * (1 - `par[3]`) +
  (x == 5) * ((`par[2]` + `par[4]` - 2) * sqrt(10)) +
  (x == 6) * ((`par[2]` - `par[4]`) * sqrt(0.1))

testnls<- try(nls(frm, data=data.frame(x = 1:6),
  start= c(`par[1]` = -3, `par[2]` = -1, `par[3]` = -3, `par[4]` = -1),
  trace=TRUE, control=list(maxiter=100)))
```

```
## 19192.    (3.68e+01): par = (-3 -1 -3 -1)
## 619.62    (8.22e+00): par = (-1.6622 0.97779 -1.6622 0.97785)
## 24.676    (1.45e+00): par = (-1.1204 0.96804 -1.1202 0.96818)
## 7.9570    (1.00e-01): par = (-0.98274 0.95569 -0.98234 0.95583)
## 7.8770    (2.19e-03): par = (-0.96962 0.95002 -0.96892 0.94977)
## 7.8770    (1.56e-04): par = (-0.96929 0.94969 -0.96823 0.94875)
## 7.8770    (1.63e-04): par = (-0.96949 0.95007 -0.968 0.94832)
## 7.8770    (1.90e-04): par = (-0.96973 0.95054 -0.96776 0.94785)
```

```
## 7.8770    (2.20e-04): par = (-0.97002 0.95109 -0.96747 0.94729)
## 7.8770    (2.56e-04): par = (-0.97035 0.95173 -0.96714 0.94666)
## 7.8770    (2.98e-04): par = (-0.97073 0.95247 -0.96675 0.94591)
## 7.8770    (3.46e-04): par = (-0.97118 0.95334 -0.96631 0.94505)
## 7.8770    (4.02e-04): par = (-0.97169 0.95434 -0.96579 0.94404)
## 7.8770    (4.67e-04): par = (-0.9723 0.95551 -0.96518 0.94288)
## 7.8770    (5.42e-04): par = (-0.97299 0.95686 -0.96448 0.94152)
## 7.8769    (6.30e-04): par = (-0.9738 0.95843 -0.96366 0.93995)
## 7.8769    (7.32e-04): par = (-0.97474 0.96026 -0.96271 0.93812)
## 7.8769    (8.51e-04): par = (-0.97583 0.96239 -0.9616 0.936)
## 7.8769    (9.89e-04): par = (-0.9771 0.96486 -0.96031 0.93353)
## 7.8769    (1.15e-03): par = (-0.97857 0.96772 -0.95882 0.93066)
## 7.8769    (1.34e-03): par = (-0.98028 0.97106 -0.95707 0.92733)
## 7.8769    (1.55e-03): par = (-0.98225 0.97493 -0.95504 0.92345)
## 7.8768    (1.81e-03): par = (-0.98455 0.97943 -0.95268 0.91894)
## 7.8768    (2.10e-03): par = (-0.98721 0.98466 -0.94992 0.91371)
## 7.8767    (2.44e-03): par = (-0.99029 0.99075 -0.9467 0.90762)
## 7.8766    (2.84e-03): par = (-0.99387 0.99783 -0.94294 0.90053)
## 7.8764    (3.31e-03): par = (-0.99801 1.0061 -0.93855 0.89228)
## 7.8763    (3.86e-03): par = (-1.0028 1.0157 -0.93341 0.88268)
## 7.8760    (4.51e-03): par = (-1.0084 1.0268 -0.92739 0.87148)
## 7.8757    (5.28e-03): par = (-1.0148 1.0399 -0.92031 0.85842)
## 7.8752    (6.19e-03): par = (-1.0223 1.0551 -0.91197 0.84316)
## 7.8745    (7.28e-03): par = (-1.031 1.0729 -0.9021 0.82528)
## 7.8736    (8.61e-03): par = (-1.0412 1.0938 -0.89038 0.80429)
## 7.8723    (1.02e-02): par = (-1.053 1.1185 -0.87637 0.77954)
## 7.8706    (1.23e-02): par = (-1.0668 1.1477 -0.85948 0.75021)
## 7.8680    (1.50e-02): par = (-1.083 1.1825 -0.8389 0.71521)
## 7.8641    (1.85e-02): par = (-1.1023 1.2244 -0.81342 0.67301)
## 7.8583    (2.35e-02): par = (-1.1254 1.2756 -0.78122 0.62144)
## 7.8488    (3.11e-02): par = (-1.1535 1.3393 -0.73931 0.55719)
## 7.8327    (4.37e-02): par = (-1.1885 1.4207 -0.68233 0.47497)
## 7.8025    (6.79e-02): par = (-1.2334 1.5286 -0.59965 0.36576)
## 7.7462    (1.27e-01): par = (-1.2935 1.6789 -0.46652 0.21351)
## 7.6604    (1.94e-01): par = (-1.3346 1.7869 -0.33904 0.10384)
## 7.6142    (3.74e-01): par = (-1.3821 1.9155 -0.12738 -0.026623)
## 7.5415    (5.59e-01): par = (-1.3949 1.9516 0.069999 -0.062512)
## 7.3436    (6.66e-01): par = (-1.3849 1.924 0.20623 -0.033426)
## 7.1157    (7.70e-01): par = (-1.3598 1.8548 0.35464 0.038608)
## 6.8262    (8.44e-01): par = (-1.3186 1.7428 0.50135 0.15459)
## 6.4578    (8.79e-01): par = (-1.2645 1.6006 0.63512 0.30162)
## 6.2472    (9.84e-01): par = (-1.14 1.2871 0.86884 0.62518)
## 5.5336    (9.75e-01): par = (-0.99934 0.97125 1.0369 0.95044)
## 5.4113    (1.19e+00): par = (-0.70883 0.40788 1.285 1.5288)
## 5.1175    (1.53e+00): par = (-0.39265 0.010715 1.4181 1.9333)
## 4.5308    (1.85e+00): par = (-0.19561 -0.10623 1.453 2.0521)
## 4.4240    (2.85e+00): par = (0.035539 -0.1588 1.4667 2.1074)
## 3.9614    (3.66e+00): par = (0.16047 -0.1293 1.4556 2.0804)
## 3.5060    (4.87e+00): par = (0.27989 -0.071143 1.4351 2.0255)
## 3.2871    (1.02e+01): par = (0.49442 0.086932 1.3795 1.8748)
## 3.1866    (5.21e+01): par = (0.79662 0.46502 1.243 1.5124)
## 0.45180   (1.71e+03): par = (1.0212 0.99253 1.0262 1.0063)
## 5.9748e-05 (1.45e+05): par = (1.0002 1.0001 1.0003 0.99993)
## 1.0997e-12 (6.29e+08): par = (1 1 1 1)
```

```
## 9.3750e-28 (7.86e+02): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## 0.0000    (NaN): par = (1 1 1 1)
## Error in nls(frm, data = data.frame(x = 1:6), start = c(`par[1]` = -3,  :
##   number of iterations exceeded maximum of 100
```

```
testnlxb<- try(nlxb(frm, data=data.frame(x = 1:6),
              start= c(`par[1]` = -3, `par[2]` = -1, `par[3]` = -3, `par[4]` = -1),
              trace=TRUE))
```

```
## Error in deriv.default(residexpr, names(pvec)) :
##   Function '`==`' is not in the derivatives table
```

```
testnlxbn<- try(nlxb(frm, data=data.frame(x = 1:6),
              start= c(`par[1]` = -3, `par[2]` = -1, `par[3]` = -3, `par[4]` = -1),
              trace=TRUE, control=list(japprox="jacentral")))
```

```
## Using approximation  jacentral
##       is.character(ctrl$japprox)= TRUE
```

```
## No backtrack
## 00lamda: 1e-04  SS= 19192 ( NA ) at  par[1] = -3  par[2] = -1  par[3] = -3  par[4] = -1 f/j  1 / 0
## <<lamda: 4e-05  SS= 619.62 ( 0.7222 ) at  par[1] = -1.6622  par[2] = 0.97778  par[3] = -1.6622  par[4
## <<lamda: 1.6e-05  SS= 24.676 ( 0.71949 ) at  par[1] = -1.1204  par[2] = 0.96804  par[3] = -1.1202  pa
## <<lamda: 6.4e-06  SS= 7.957 ( 0.58492 ) at  par[1] = -0.98274  par[2] = 0.95569  par[3] = -0.98234  p
## <<lamda: 2.56e-06  SS= 7.877 ( 0.067351 ) at  par[1] = -0.96962  par[2] = 0.95002  par[3] = -0.96892
## <<lamda: 1.024e-06  SS= 7.877 ( 0.0014049 ) at  par[1] = -0.96929  par[2] = 0.94969  par[3] = -0.9682
## <<lamda: 4.096e-07  SS= 7.877 ( 0.0001389 ) at  par[1] = -0.96949  par[2] = 0.95007  par[3] = -0.968
## <<lamda: 1.6384e-07  SS= 7.877 ( 0.00015204 ) at  par[1] = -0.96973  par[2] = 0.95054  par[3] = -0.96
## <<lamda: 6.5536e-08  SS= 7.877 ( 0.00017634 ) at  par[1] = -0.97002  par[2] = 0.95109  par[3] = -0.96
## <<lamda: 2.6214e-08  SS= 7.877 ( 0.00020487 ) at  par[1] = -0.97035  par[2] = 0.95173  par[3] = -0.96
## <<lamda: 1.0486e-08  SS= 7.877 ( 0.00023803 ) at  par[1] = -0.97073  par[2] = 0.95247  par[3] = -0.96
## <<lamda: 4.1943e-09  SS= 7.877 ( 0.00027656 ) at  par[1] = -0.97118  par[2] = 0.95334  par[3] = -0.96
## <<lamda: 1.6777e-09  SS= 7.877 ( 0.00032133 ) at  par[1] = -0.97169  par[2] = 0.95434  par[3] = -0.96
## <<lamda: 6.7109e-10  SS= 7.877 ( 0.00037335 ) at  par[1] = -0.9723  par[2] = 0.95551  par[3] = -0.965
## <<lamda: 2.6844e-10  SS= 7.877 ( 0.0004338 ) at  par[1] = -0.97299  par[2] = 0.95686  par[3] = -0.964
## <<lamda: 1.0737e-10  SS= 7.8769 ( 0.00050404 ) at  par[1] = -0.9738  par[2] = 0.95843  par[3] = -0.96
## <<lamda: 4.295e-11  SS= 7.8769 ( 0.00058568 ) at  par[1] = -0.97474  par[2] = 0.96026  par[3] = -0.96
## <<lamda: 1.718e-11  SS= 7.8769 ( 0.00068056 ) at  par[1] = -0.97583  par[2] = 0.96239  par[3] = -0.96
## <<lamda: 6.8719e-12  SS= 7.8769 ( 0.00079085 ) at  par[1] = -0.9771  par[2] = 0.96485  par[3] = -0.96
## <<lamda: 2.7488e-12  SS= 7.8769 ( 0.00091908 ) at  par[1] = -0.97857  par[2] = 0.96772  par[3] = -0.9
## <<lamda: 1.0995e-12  SS= 7.8769 ( 0.0010682 ) at  par[1] = -0.98028  par[2] = 0.97106  par[3] = -0.95
## <<lamda: 4.398e-13  SS= 7.8769 ( 0.0012417 ) at  par[1] = -0.98225  par[2] = 0.97493  par[3] = -0.955
## <<lamda: 1.7592e-13  SS= 7.8768 ( 0.0014435 ) at  par[1] = -0.98455  par[2] = 0.97943  par[3] = -0.95
## <<lamda: 7.0369e-14  SS= 7.8768 ( 0.0016785 ) at  par[1] = -0.98721  par[2] = 0.98466  par[3] = -0.94
## <<lamda: 2.8147e-14  SS= 7.8767 ( 0.0019524 ) at  par[1] = -0.99029  par[2] = 0.99075  par[3] = -0.94
## <<lamda: 1.1259e-14  SS= 7.8766 ( 0.0022718 ) at  par[1] = -0.99387  par[2] = 0.99783  par[3] = -0.94
## <<lamda: 4.5036e-15  SS= 7.8764 ( 0.0026448 ) at  par[1] = -0.99801  par[2] = 1.0061  par[3] = -0.938
## <<lamda: 1.8014e-15  SS= 7.8763 ( 0.0030813 ) at  par[1] = -1.0028  par[2] = 1.0157  par[3] = -0.9334
## <<lamda: 7.2058e-16  SS= 7.876 ( 0.0035933 ) at  par[1] = -1.0084  par[2] = 1.0268  par[3] = -0.92739
## <<lamda: 2.8823e-16  SS= 7.8757 ( 0.0041957 ) at  par[1] = -1.0148  par[2] = 1.0399  par[3] = -0.9203
## <<lamda: 1.1529e-16  SS= 7.8752 ( 0.0049077 ) at  par[1] = -1.0223  par[2] = 1.0551  par[3] = -0.9119
## <<lamda: 4.6117e-17  SS= 7.8745 ( 0.0057541 ) at  par[1] = -1.031  par[2] = 1.0729  par[3] = -0.9021
## <<lamda: 1.8447e-17  SS= 7.8736 ( 0.0067682 ) at  par[1] = -1.0412  par[2] = 1.0938  par[3] = -0.8903
## <<lamda: 7.3787e-18  SS= 7.8723 ( 0.0079962 ) at  par[1] = -1.053  par[2] = 1.1185  par[3] = -0.87637
## <<lamda: 2.9515e-18  SS= 7.8706 ( 0.0095043 ) at  par[1] = -1.0668  par[2] = 1.1477  par[3] = -0.8594
## <<lamda: 1.1806e-18  SS= 7.868 ( 0.011391 ) at  par[1] = -1.083  par[2] = 1.1825  par[3] = -0.8389  p
## <<lamda: 4.7224e-19  SS= 7.8641 ( 0.013812 ) at  par[1] = -1.1023  par[2] = 1.2244  par[3] = -0.81342
## <<lamda: 1.8889e-19  SS= 7.8583 ( 0.017022 ) at  par[1] = -1.1254  par[2] = 1.2756  par[3] = -0.78122
## <<lamda: 7.5558e-20  SS= 7.8488 ( 0.021473 ) at  par[1] = -1.1535  par[2] = 1.3393  par[3] = -0.73931
## <<lamda: 3.0223e-20  SS= 7.8327 ( 0.028022 ) at  par[1] = -1.1885  par[2] = 1.4207  par[3] = -0.68234
## <<lamda: 1.2089e-20  SS= 7.8025 ( 0.03848 ) at  par[1] = -1.2334  par[2] = 1.5286  par[3] = -0.59965
## <<lamda: 4.8357e-21  SS= 7.7462 ( 0.057168 ) at  par[1] = -1.2935  par[2] = 1.6788  par[3] = -0.46654
## <<lamda: 0.088818  SS= 7.638 ( 0.079082 ) at  par[1] = -1.3491  par[2] = 1.8261  par[3] = -0.28355  p
## <<lamda: 0.35527  SS= 7.4545 ( 0.13708 ) at  par[1] = -1.3836  par[2] = 1.9221  par[3] = -0.043038  p
## <<lamda: 1.4211  SS= 6.9097 ( 0.20552 ) at  par[1] = -1.3775  par[2] = 1.906  par[3] = 0.1662  par[4]
## <<lamda: 5.6843  SS= 6.6077 ( 0.16179 ) at  par[1] = -1.3695  par[2] = 1.8836  par[3] = 0.22365  par
## <<lamda: 2.2737  SS= 6.2637 ( 0.14847 ) at  par[1] = -1.3463  par[2] = 1.8202  par[3] = 0.36595  par
## <<lamda: 0.90949  SS= 5.8217 ( 0.23149 ) at  par[1] = -1.2786  par[2] = 1.6382  par[3] = 0.6048  par
## <<lamda: 0.3638  SS= 5.2107 ( 0.29845 ) at  par[1] = -1.1188  par[2] = 1.2339  par[3] = 0.89683  par
## <<lamda: 0.14552  SS= 4.7354 ( 0.33099 ) at  par[1] = -0.82861  par[2] = 0.61006  par[3] = 1.1961  pa
## <<lamda: 0.58208  SS= 3.0093 ( 0.36399 ) at  par[1] = -0.66763  par[2] = 0.42847  par[3] = 1.2339  pa
## <<lamda: 0.23283  SS= 2.8741 ( 0.26092 ) at  par[1] = -0.39404  par[2] = 0.089666  par[3] = 1.3679  p
## <<lamda: 0.93132  SS= 2.0754 ( 0.41227 ) at  par[1] = -0.19541  par[2] = 0.0080271  par[3] = 1.3914
```

```
## <<lamda: 3.7253  SS= 1.7695 ( 0.26762 ) at  par[1] = -0.10339  par[2] = 0.008526  par[3] = 1.3949  pa
## <<lamda: 1.4901  SS= 1.5047 ( 0.22996 ) at  par[1] = 0.095894  par[2] = -0.022696  par[3] = 1.4051  p
## <<lamda: 5.9605  SS= 1.2694 ( 0.25391 ) at  par[1] = 0.14995  par[2] = 0.017241  par[3] = 1.4016  pa
## <<lamda: 2.3842  SS= 1.0532 ( 0.18813 ) at  par[1] = 0.27319  par[2] = 0.059706  par[3] = 1.3892  pa
## <<lamda: 0.95367  SS= 0.82749 ( 0.23832 ) at  par[1] = 0.46738  par[2] = 0.17946  par[3] = 1.3497  pa
## <<lamda: 0.38147  SS= 0.51684 ( 0.33553 ) at  par[1] = 0.6751  par[2] = 0.41126  par[3] = 1.2643  pa
## <<lamda: 0.15259  SS= 0.20247 ( 0.34051 ) at  par[1] = 0.85441  par[2] = 0.69729  par[3] = 1.1475  pa
## <<lamda: 0.061035  SS= 0.028137 ( 0.29007 ) at  par[1] = 0.95967  par[2] = 0.90974  par[3] = 1.0487
## <<lamda: 0.024414  SS= 0.00055808 ( 0.10844 ) at  par[1] = 0.9941  par[2] = 0.98703  par[3] = 1.0073
## <<lamda: 0.0097656  SS= 9.0254e-07 ( 0.016403 ) at  par[1] = 0.9996  par[2] = 0.99918  par[3] = 1.000
## <<lamda: 0.0039062  SS= 4.3134e-10 ( 0.0007607 ) at  par[1] = 0.99999  par[2] = 0.99998  par[3] = 1
## <<lamda: 0.0015625  SS= 4.917e-14 ( 2.0305e-05 ) at  par[1] = 1  par[2] = 1  par[3] = 1  par[4] = 1
## <<lamda: 0.000625  SS= 9.1933e-19 ( 2.1882e-07 ) at  par[1] = 1  par[2] = 1  par[3] = 1  par[4] = 1
```

```
testnlsLM<- try(nlsLM(frm, data=data.frame(x = 1:6),
                start= c(`par[1]` = -3, `par[2]` = -1, `par[3]` = -3, `par[4]` = -1),
                trace=TRUE, control=list(maxiter=100)))
```

```
## It.    0, RSS =      19192, Par. =         -3         -1         -3         -1
## It.    1, RSS =     619.62, Par. =   -1.66222   0.977785   -1.66215   0.977845
## It.    2, RSS =    24.6758, Par. =   -1.12038   0.968035   -1.12019   0.968185
## It.    3, RSS =    7.95701, Par. =  -0.982743   0.955689  -0.982343   0.955827
## It.    4, RSS =    7.87701, Par. =  -0.969625   0.950021  -0.968922   0.949765
## It.    5, RSS =    7.87697, Par. =  -0.969295   0.949687   -0.96823   0.948754
## It.    6, RSS =    7.87697, Par. =  -0.969489   0.950068  -0.968002    0.94832
## It.    7, RSS =    7.87697, Par. =  -0.969733   0.950541  -0.967756   0.947845
## It.    8, RSS =    7.87696, Par. =  -0.970018   0.951092  -0.967471   0.947295
## It.    9, RSS =    7.87696, Par. =  -0.970348   0.951732   -0.96714   0.946655
## It.   10, RSS =    7.87696, Par. =  -0.970732   0.952475  -0.966755   0.945912
## It.   11, RSS =    7.87696, Par. =  -0.971177   0.953338  -0.966307   0.945048
## It.   12, RSS =    7.87696, Par. =  -0.971695   0.954342  -0.965786   0.944045
## It.   13, RSS =    7.87696, Par. =  -0.972296   0.955507  -0.965181   0.942879
## It.   14, RSS =    7.87695, Par. =  -0.972993   0.956861  -0.964478   0.941525
## It.   15, RSS =    7.87695, Par. =  -0.973803   0.958435   -0.96366   0.939951
## It.   16, RSS =    7.87694, Par. =  -0.974743   0.960263  -0.962709   0.938123
## It.   17, RSS =    7.87693, Par. =  -0.975834   0.962387  -0.961602   0.935998
## It.   18, RSS =    7.87692, Par. =  -0.977101   0.964855  -0.960315   0.933529
## It.   19, RSS =     7.8769, Par. =   -0.97857   0.967723  -0.958817    0.93066
## It.   20, RSS =    7.87688, Par. =  -0.980275   0.971055  -0.957073   0.927325
## It.   21, RSS =    7.87685, Par. =  -0.982253   0.974928  -0.955042    0.92345
## It.   22, RSS =    7.87681, Par. =  -0.984547    0.97943  -0.952676   0.918945
## It.   23, RSS =    7.87675, Par. =  -0.987207   0.984663  -0.949918   0.913707
## It.   24, RSS =    7.87668, Par. =  -0.990291   0.990748    -0.9467   0.907616
## It.   25, RSS =    7.87658, Par. =  -0.993867   0.997826  -0.942944   0.900531
## It.   26, RSS =    7.87644, Par. =  -0.998011    1.00606  -0.938553   0.892284
## It.   27, RSS =    7.87625, Par. =   -1.00282    1.01565  -0.933414    0.88268
## It.   28, RSS =      7.876, Par. =   -1.00839    1.02682  -0.927389   0.871486
## It.   29, RSS =    7.87565, Par. =   -1.01484    1.03986   -0.92031   0.858424
## It.   30, RSS =    7.87518, Par. =   -1.02234    1.05508  -0.911969   0.843159
## It.   31, RSS =    7.87452, Par. =   -1.03104    1.07291  -0.902105   0.825284
## It.   32, RSS =    7.87362, Par. =   -1.04117    1.09383  -0.890386   0.804292
## It.   33, RSS =    7.87235, Par. =   -1.05298    1.11848  -0.876375   0.779544
## It.   34, RSS =    7.87055, Par. =   -1.06679    1.14767  -0.859485   0.750217
## It.   35, RSS =    7.86796, Par. =   -1.08304    1.18248  -0.838898   0.715212
## It.   36, RSS =    7.86413, Par. =   -1.10231    1.22442  -0.813421   0.673013
```

```
## It.   37, RSS =    7.85826, Par. =   -1.12539    1.27561  -0.781225   0.621441
## It.   38, RSS =    7.84884, Par. =   -1.15349    1.33931  -0.739312   0.557194
## It.   39, RSS =    7.83265, Par. =   -1.18846     1.4207  -0.682338   0.474981
## It.   40, RSS =    7.80255, Par. =   -1.23339    1.52863  -0.599656   0.365776
## It.   41, RSS =    7.74624, Par. =   -1.29351    1.67884  -0.466541   0.213528
## It.   42, RSS =    7.63602, Par. =    -1.3238    1.76075  -0.358081   0.130412
## It.   43, RSS =      7.546, Par. =   -1.37731    1.90331  -0.138102 -0.0147591
## It.   44, RSS =    7.14749, Par. =   -1.38383    1.92366  0.0924426 -0.0305445
## It.   45, RSS =    6.69306, Par. =   -1.36486    1.87016   0.180756  0.0369628
## It.   46, RSS =    6.28704, Par. =   -1.33733    1.79561   0.360371    0.10984
## It.   47, RSS =    6.22658, Par. =   -1.24142    1.53916   0.692721   0.379469
## It.   48, RSS =    4.66718, Par. =   -1.13433    1.28049   0.823399   0.666767
## It.   49, RSS =    4.33323, Par. =  -0.923112   0.813502    1.09706    1.13376
## It.   50, RSS =    3.59357, Par. =  -0.641045   0.337289     1.2851    1.61924
## It.   51, RSS =    3.29269, Par. =  -0.315928  0.000515302    1.40285    1.95655
## It.   52, RSS =    1.93188, Par. =  -0.135019 -0.00903142    1.40453    1.97433
## It.   53, RSS =     1.6448, Par. = -0.0399549 -0.00215784    1.40261    1.96884
## It.   54, RSS =     1.3786, Par. =   0.134289 -0.00689825    1.40387    1.97247
## It.   55, RSS =    1.10057, Par. =   0.232553   0.048798    1.38708     1.9247
## It.   56, RSS =   0.856639, Par. =   0.428938   0.149378    1.35166    1.82664
## It.   57, RSS =   0.500123, Par. =   0.613057   0.344464    1.28285     1.6413
## It.   58, RSS =   0.271392, Par. =   0.684328   0.464694    1.23733     1.5287
## It.   59, RSS =   0.142606, Par. =   0.832837   0.672593      1.153    1.32228
## It.   60, RSS =  0.0475829, Par. =   0.965099   0.914157    1.04693    1.08479
## It.   61, RSS = 0.00056213, Par. =    1.00072    1.00018    1.00095   0.999783
## It.   62, RSS = 9.61727e-11, Par. =          1          1          1          1
## It.   63, RSS = 2.22262e-14, Par. =          1          1          1          1
## It.   64, RSS =          0, Par. =          1          1          1          1
```

The results above for `nlsr::nlxb()` accord with the accepted solution, but a large number of iterations are used. The Wood problem is, however, known to be difficult.

Both `nls()` and `minpack.lm::nlsLM()` required the specification of more than the default number of "iterations" (essentially Jacobian evaluations, which `nlsr` limits with the control parameter `jemax` for which the default is a very aggressive 5000). Note that `nls()` did NOT terminate except on this limit. I have not determined, as yet, the precise cause of this.

Ongoing efforts

The examples above show how to use package `nlsr` as it exists at the beginning of February 2024. 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

Hartley, H. O. 1961. "The Modified Gauss-Newton Method for Fitting of Nonlinear Regression Functions by Least Squares." *Technometrics* 3: 269–80.

John C Nash, and Duncan Murdoch. 2019. *nlsr: Functions for Nonlinear Least Squares Solutions.*

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.

Moré, Jorge J., Burton S. Garbow, and Kenneth E. Hillstrom. 1981. "Testing Unconstrained Optimization Software." *J-Toms* 7 (1): 17–41.

Nash, J. C. 1979. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation.* Book. Bristol: Hilger: Bristol.

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

Nash, John C., and Arkajyoti Bhattacharjee. 2022. "Refactoring the 'nls()' function in R." https://github.com/nashjc/RNonlinearLS/blob/main/RefactoringNLS/RefactoringNLS.pdf.

Nash, John C., and Mary Walker-Smith. 1987. *Nonlinear Parameter Estimation: An Integrated System in BASIC.* Book. New York: Marcel Dekker.

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

Rosenbrock, H. H. 1960. "An Automatic Method for Finding the Greatest or Least Value of a Function." *The Computer Journal* 3: 175–84.