A comparison of nlsr::nlxb with nls and minpack::nlsLM

John C. Nash

2021-11-22

R has several tools for estimating nonlinear models and minimizing sums of squares functions. Sometimes we talk of **nonlinear regression** and at other times of **minimizing a sum of squares function**. Many workers conflate these two tasks. Here some of the differences are highlighted by comparing the tools from the package `nlsr` (Nash and Murdoch (2019)), particularly function `nlxb()` with those from base-R `nls()` and the `nlsLM` function of package `minpack.lm` (Elzhov et al. (2012)).

Principal differences

The main differences in the tools relate to the following features:

- the way in which derivative information is computed for the Jacobian of the modelling function
- the use of a Marquardt stabilization for solution of the linearized least squares problem at each iteration
- details of the criterion used to terminate the iteration
- the structure of the output of the tools
- how models are predicted for new data.

Derivative information

`nlsr::nlxb()` attempts to use symbolic and algorithmic tools to obtain the derivatives of the model expression that are needed for the **Jacobian** matrix that is used in creating a linearized sub-problem at each iteration of an attempted solution of the minimization of the sum of squared residuals. `nls()` and `minpack.lm::nlsLM()` use a very simple forward-difference approximation for the partial derivatives. For the i'th partial derivative of the modelling function with respect to parameter xx, they use (in `C`)

```
delta = (xx == 0) ? eps : xx*eps;
```

and

```
REAL(gradient)[start + k] = rDir[i] * (REAL(ans_del)[k] - REAL(ans)[k])/delta;
```

where

```
double eps = sqrt(DOUBLE_EPS), *rDir;
```

and where `DOUBLE_EPS` in Constants.h in the **R** source code refers to `DBL_EPSILON` in float.h in most C compilers, e.g.,

```
#define DBL_EPSILON 2.2204460492503131e-16
```

Thus `delta` is of the order of 1.490116e-08.

Forward difference approximations are less accurate than central differences, and both are subject to numerical error when the modelling function is "flat", so that there is a large amount of digit cancellation in the subtraction necessary to compute the derivative approximation.

`minpack.lm::nlsLM` uses the same derivatives as far as I can determine. The loss of information compared to the analytic or algorithmic derivatives of `nlsr::nlxb()` is important in that it can lead to Jacobian matrices that are computationally singular, where `nls()` will stop with "singular gradient". (It is actually the

Jacobian which is singular here, and I will stay with that terminology.) `minpack.lm::nlsLM()` may fail to get started if the initial Jacobian is singular, but is less susceptible in general, as described in the sub-section on Marquardt stabilization which follows.

Consequences of different derivative computations While readers might expect that the precise derivative information of `nlsr::nlxb()` would mean a faster solution, this is quite often not the case. Approximate derivatives may allow faster approach to the solution by "ironing out" wrinkles in the function surface. In my opinion, the main advantage of precise derivative information is in testing that we actually have arrived at a solution.

There are even some cases where the approximation may be helpful, though users may not realize the potential danger. Thanks to Karl Schilling for an example of modelling with the function

```
a * (x ^ b)
```

where `x` is our data and we wish to estimate `a` and `b`. Now the partial derivative of this function w.r.t. `b` is

```r
partialderiv <- D(expression(a * (x ^ b)),"b")
print(partialderiv)
```

```
## a * (x^b * log(x))
```

The danger here is that we may have data values `x = 0`, in which case the **derivative** is not defined, though the model can still be evaluated. Thus `nlsr::nlxb()` will not compute a solution, while `nls()` and `minpack.lm::nlsLM()` will generally proceed. A workaround is to provide a very small value instead of zero for the data, though I find this inelegant. Another approach is to drop the offending element of the data, though this risks altering the model estimated. A proper treatment might be to develop the limit of the derivative as the data value goes to zero, but finding general software that can detect and deal with this is a large project.

Marquardt stabilization

All three of the R functions under consideration try to minimize a sum of squares. If the model is provided in the form

```
y ~ (some expression)
```

then the residuals are computed by evaluating the difference between `(some expression)` and `y`. My own preference, and that of K F Gauss, is to use `(some expression) - y`. This is to avoid having to be concerned with the negative sign – the derivative of the residual defined in this way is the same as the derivative of the modelling function, and we avoid the chance of a sign error. The Jacobian matrix is made up of elements where element `i, j` is the partial derivative of residual `i` w.r.t. parameter `j`.

`nls()` attempts to minimize a sum of squared residuals by a Gauss-Newton method. If we compute a Jacobian matrix `J` and a vector of residuals `r` from a vector of parameters `x`, then we can define a linearized problem

$$J^T J\delta = -J^T r$$

This leads to an iteration where, from a set of starting parameters `x0`, we compute

$$x_{i+1} = x_i + \delta$$

This is commonly modified to use a step factor `step`

$$x_{i+1} = x_i + step * \delta$$

It is in the mechanisms to choose the size of `step` and to decide when to terminate the iteration that Gauss-Newton methods differ. Indeed, though I have tried several times, I find the very convoluted code behind `nls()` very difficult to decipher. Unfortunately, its authors have (at 2018 as far as I am aware) all ceased to maintain the code.

Both `nlsr::nlxb()` and `minpack.lm::nlsLM` use a Levenberg-Marquardt stabilization of the iteration. (Marquardt (1963), Levenberg (1944)), solving

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

where D is some diagonal matrix and lambda is a number of modest size initially. 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.

As for Gauss-Newton methods, the details of how to start, adjust and terminate the iteration lead to many variants, increased by different possibilities for specifying D. See Nash (1979). There are also a number of ways to solve the stabilized Gauss-Newton equations, some of which do not require the explicit $J^T J$ matrix.

Criterion used to terminate the iteration

`nls()` and `nlsr` use a form of the relative offset convergence criterion, Bates and Watts (1981). `minpack.lm` uses a somewhat different and more complicated set of tests. Unfortunately, the relative offset criterion as implemented in `nls()` is unsuited to problems where the residuals can be zero. There are ways to work around the difficulties, and `nlsr` has used one approach. See *An illustrative nonlinear regression problem* below.

Output of the modelling functions

`nls()` and `nlsLM()` return the same solution structure. Let us examine this for one of our example results (we will choose one that does NOT have small residuals, so that all the functions "work").

```
str(nlsy0t0ax)
```

```
## List of 6
##  $ m          :List of 16
##   ..$ resid     :function ()
##   ..$ fitted    :function ()
##   ..$ formula   :function ()
##   ..$ deviance  :function ()
##   ..$ lhs       :function ()
##   ..$ gradient  :function ()
##   ..$ conv      :function ()
##   ..$ incr      :function ()
##   ..$ setVarying:function (vary = rep_len(TRUE, np))
##   ..$ setPars   :function (newPars)
##   ..$ getPars   :function ()
##   ..$ getAllPars:function ()
##   ..$ getEnv    :function ()
##   ..$ trace     :function ()
##   ..$ Rmat      :function ()
##   ..$ predict   :function (newdata = list(), qr = FALSE)
##   ..- attr(*, "class")= chr "nlsModel"
##  $ convInfo   :List of 5
##   ..$ isConv     : logi TRUE
##   ..$ finIter    : int 6
##   ..$ finTol     : num 4.75e-10
##   ..$ stopCode   : int 0
##   ..$ stopMessage: chr "converged"
##  $ data       : symbol edta
##  $ call       : language nls(formula = y1 ~ a * (t0a^b), data = edta, start = start1, control = list
##  $ dataClasses: Named chr "numeric"
##   ..- attr(*, "names")= chr "t0a"
##  $ control    :List of 7
##   ..$ maxiter    : num 10000
##   ..$ tol        : num 1e-05
##   ..$ minFactor  : num 0.000977
##   ..$ printEval  : logi FALSE
##   ..$ warnOnly   : logi FALSE
##   ..$ scaleOffset: num 0
##   ..$ nDcentral  : logi FALSE
##  - attr(*, "class")= chr "nls"
```

The `minpack.lm::nlsLM` output has the same structure, which could be revealed by the R command `str(nlsLMy1t0a)`. Note that this structure has a lot of special functions in the sub-list `m`. By contrast, the `nlsr()` output is much less flamboyant. There are, in fact, no functions as part of the structure.

```
str(nlsry1t0a)
```

```
## List of 11
##  $ resid       : num [1:20] 4.00e-05 -2.03e-09 -1.70e-09 -1.41e-09 -1.16e-09 ...
##   ..- attr(*, "gradient")= num [1:20, 1:2] 0.00001 1 1.18921 1.31607 1.41421 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "a" "b"
##  $ jacobian    : num [1:20, 1:2] 0.00001 1 1.18921 1.31607 1.41421 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "a" "b"
##  $ feval       : num 8
##  $ jeval       : num 7
##  $ coefficients: Named num [1:2] 4 0.25
##   ..- attr(*, "names")= chr [1:2] "a" "b"
##  $ ssquares    : num 1.6e-09
##  $ lower       : num [1:2] -Inf -Inf
##  $ upper       : num [1:2] Inf Inf
##  $ maskidx     : int(0)
##  $ weights     : NULL
##  $ formula     :Class 'formula'  language y1 ~ a * (t0a^b)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  - attr(*, "class")= chr "nlsr"
```

Which of these approaches is “better” can be debated. My preference is for the results of optimization computations to be essentially data, including messages, though some tools within some of my packages will return functions for specific reasons, e.g., to return a function from an expression. However, I prefer to use specified functions such as `predict.nlsr()` below to obtain predictions. I welcome comment and discussion, as this is not, in my view, a closed topic.

Prediction

Let us predict our models at the mean of the data. Because `nlxb()` returns a different structure from that found by `nls()` and `nlsLM()` the code for `predict()` for an object from `nlsr` is different. `minpack.lm` uses `predict.nls` since the output structure of the modelling step is equivalent to that from `nls()`.

```
nudta <- colMeans(edta)
predict(nlsy0t0ax, newdata=nudta)
```

```
## [1] 7.0225
```

```
predict(nlsLMy1t0a, newdata=nudta)
```

```
## [1] 7.0225
```

```
predict(nlsry1t0a, newdata=nudta)
```

```
## [1] 7.0225
## attr(,"class")
## [1] "predict.nlsr"
## attr(,"pkgname")
## [1] "nlsr"
```

An illustrative nonlinear regression problem

So we can illustrate some of the issues, let us create some example data for a seemingly straightforward computational problem.

```
# Here we set up an example problem with data
# Define independent variable
t0 <- 0:19
t0a<-t0
t0a[1]<-1e-20 # very small value
# Drop first value in vectors
t0t<-t0[-1]
y1 <- 4 * (t0^0.25)
y1t<-y1[-1]
n <- length(t0)
fuzz <- rnorm(n)
range <- max(y1)-min(y1)
## add some "error" to the dependent variable
y1q <- y1 + 0.2*range*fuzz
edta <- data.frame(t0=t0, t0a=t0a, y1=y1, y1q=y1q)
edtat <- data.frame(t0t=t0t, y1t=y1t)
```

Let us try this example modelling `y0` against `t0`. Note that this is a zero-residual problem, so `nls()` should complain or fail, which it appears to do but by exceeding the iteration limit, which is not very communicative of the underlying issue. The `nls()` documentation warns

"Warning

Do not use nls on artificial "zero-residual" data."

It goes on to recommend that users add "error" to the data to avoid such problems. I feel this is a very unsatisfactory kludge. It is NOT due to a genuine mathematical issue, but due to the relative offset convergence criterion used to terminate the method. In October 2020, I suggested a patch for nls() to R-core that it seems will become part of base R eventually. This patch allows the user to specify a parameter, tentatively named `convTestAdd` with a zero default value, in `nls.control()`. (This allows existing examples using `nsl()` to function without change.) A small positive value for this control parameter avoids a zero divided by zero issue in the relative offset convergence test in `nls()` used to terminate iterations. This adjustment to the convergence test has been in `nlsr` since its creation.

Here is the output.

nls

```
cprint <- function(obj){
   # print object if it exists
  sobj<-deparse(substitute(obj))
  if (exists(sobj)) {
      print(obj)
  } else {
      cat(sobj," does not exist\n")
  }
#  return(NULL)
}
start1 <- c(a=1, b=1)
try(nlsy0t0 <- nls(formula=y1~a*(t0^b), start=start1, data=edta))
```

```
## Error in nls(formula = y1 ~ a * (t0^b), start = start1, data = edta) :
##   number of iterations exceeded maximum of 50
```

```
cprint(nlsy0t0)
```

```
## nlsy0t0  does not exist
```

```
# Since this fails to converge, let us increase the maximum iterations
try(nlsy0t0x <- nls(formula=y1~a*(t0^b), start=start1, data=edta,
                    control=nls.control(maxiter=10000)))
```

```
## Error in nls(formula = y1 ~ a * (t0^b), start = start1, data = edta, control = nls.control(maxiter =
##   number of iterations exceeded maximum of 10000
```

```
cprint(nlsy0t0x)
```

```
## nlsy0t0x  does not exist
```

```
try(nlsy0t0ax <- nls(formula=y1~a*(t0a^b), start=start1, data=edta,
                     control=nls.control(maxiter=10000)))
cprint(nlsy0t0ax)
```

```
## Nonlinear regression model
##   model: y1 ~ a * (t0a^b)
##    data: edta
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 1.6e-09
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 4.75e-10
```

```
try(nlsy0t0t <- nls(formula=y1t~a*(t0t^b), start=start1, data=edtat))
```

```
## Error in nls(formula = y1t ~ a * (t0t^b), start = start1, data = edtat) :
##   number of iterations exceeded maximum of 50
```

```
cprint(nlsy0t0t)
```

```
## nlsy0t0t  does not exist
```

nlsr

```
library(nlsr)
nlsry1t0 <- try(nlxb(formula=y1~a*(t0^b), start=start1, data=edta))
```

```
## Error in model2rjfun(formula, pnum, data = data) : Jacobian contains NaN
```

```
cprint(nlsry1t0)
```

```
## [1] "Error in model2rjfun(formula, pnum, data = data) : Jacobian contains NaN\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in model2rjfun(formula, pnum, data = data): Jacobian contains NaN>
```

```
nlsry1t0a <- nlxb(formula=y1~a*(t0a^b), start=start1, data=edta)
cprint(nlsry1t0a)
```

```
## nlsr object: x
## residual sumsquares =  1.6e-09  on  20 observations
##     after  7    Jacobian and  8 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                     4     4.811e-06     831443   1.02e-96  -1.548e-14       72.97
## b                  0.25     5.011e-07     498907  1.003e-92   -1.16e-13        1.95
```

```
nlsry1t0t <- nlxb(formula=y1t~a*(t0t^b), start=start1, data=edtat)
cprint(nlsry1t0t)
```

```
## nlsr object: x
## residual sumsquares =  2.1496e-29  on  19 observations
##     after  7    Jacobian and  8 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                     4     5.738e-16  6.971e+15  2.53e-260  -3.387e-14       72.97
## b                  0.25     5.977e-17  4.183e+15  1.493e-256  -3.182e-13        1.95
```

minpack.lm

```
library(minpack.lm)
nlsLMy1t0 <- nlsLM(formula=y1~a*(t0^b), start=start1, data=edta)
nlsLMy1t0
```

```
## Nonlinear regression model
##   model: y1 ~ a * (t0^b)
##    data: edta
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 0
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

```
nlsLMy1t0a <- nlsLM(formula=y1~a*(t0a^b), start=start1, data=edta)
nlsLMy1t0a
```

```
## Nonlinear regression model
##   model: y1 ~ a * (t0a^b)
##    data: edta
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 1.6e-09
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

```r
nlsLMy1t0t <- nlsLM(formula=y1t~a*(t0t^b), start=start1, data=edtat)
nlsLMy1t0t
```

```
## Nonlinear regression model
##   model: y1t ~ a * (t0t^b)
##    data: edtat
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 0
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

We have seemingly found a workaround for our difficulty, but I caution that initially I found very unsatisfactory results when I set the "very small value" to 1.0e-7. The correct approach is clearly to understand what is going on. Getting computers to provide that understanding is a serious challenge.

Problems that are NOT regressions

Some nonlinear least squares problems are NOT nonlinear regressions. That is, we do not have a formula `y ~ (some function)` to define the problem. This is another reason to use the residual in the form `(some function) - y` In many cases of interest we have no `y`.

The Brown and Dennis test problem (Moré, Garbow, and Hillstrom (1981), problem 16) is of this form. Suppose we have `m` observations, then we create a scaled index `t` which is the "data" for the function. To run the nonlinear least squares functions that use a formula, we do, however, need a "y" variable. Clearly adding zero to the residual will not change the problem, so we set the data for "y" as all zeros. Note that `nls()` and `nlsLM()` need some extra iterations to find the solution to this somewhat nasty problem.

```r
m <- 20
t <- seq(1, m) / 5
y <- rep(0,m)
library(nlsr)
library(minpack.lm)

bddata <- data.frame(t=t, y=y)
bdform <- y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
prm0 <- c(x1=25, x2=5, x3=-5, x4=-1)
fbd <-model2ssgrfun(bdform, prm0, bddata)
cat("initial sumsquares=",as.numeric(crossprod(fbd(prm0))),"\n")
```

```
## initial sumsquares= 6.2832e+13
```

```r
nlsrbd <- nlxb(bdform, start=prm0, data=bddata, trace=FALSE)
nlsrbd
```

```
## nlsr object: x
## residual sumsquares =  85822  on  20 observations
##     after  3056    Jacobian and  4279 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## x1              -11.5944        4.017      -2.886    0.01075     0.02637         176
## x2               13.2036        1.231       10.73  1.025e-08    -0.07207        28.1
## x3             -0.403442        28.08    -0.01437     0.9887   -0.002117       3.917
## x4              0.236777        39.79    0.005951     0.9953   -0.001648       1.624
```

```
nlsbd10k <- nls(bdform, start=prm0, data=bddata, trace=FALSE,
                control=nls.control(maxiter=10000))
nlsbd10k
```

```
## Nonlinear regression model
##   model: y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
##    data: bddata
##      x1      x2      x3      x4
## -11.594  13.204  -0.403   0.237
##  residual sum-of-squares: 85822
##
## Number of iterations to convergence: 855
## Achieved convergence tolerance: 9.6e-06
```

```
nlsLMbd10k <- nlsLM(bdform, start=prm0, data=bddata, trace=FALSE,
                    control=nls.lm.control(maxiter=10000, maxfev=10000))
```

```
## Warning in nls.lm(par = start, fn = FCT, jac = jac, control = control, lower =
## lower, : resetting `maxiter' to 1024!
```

```
nlsLMbd10k
```

```
## Nonlinear regression model
##   model: y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
##    data: bddata
##      x1      x2      x3      x4
## -11.592  13.203  -0.404   0.237
##  residual sum-of-squares: 85822
##
## Number of iterations to convergence: 242
## Achieved convergence tolerance: 1.49e-08
```

Let us try predicting the "residual" for some new data.

```
ndata <- data.frame(t=c(5,6), y=c(0,0))
predict(nlsLMbd10k, newdata=ndata)
```

```
## [1]   8835.3 112766.9
```

```
# now nls
predict(nlsbd10k, newdata=ndata)
```

```
## [1]   8834.9 112764.7
```

```
# now nlsr
predict(nlsrbd, newdata=ndata)
```

```
## [1]   8834.9 112764.7
## attr(,"class")
## [1] "predict.nlsr"
## attr(,"pkgname")
## [1] "nlsr"
```

We could, of course, try setting up a different formula, since the "residuals" can be computed in any way such that their absolute value is the same. Therefore we could try moving the exponential part of the function for each equation to the left hand side as in `bdf2` below. However, we discover that the parsing of the model formula for `nls()` and `nlsLM()` fails for this formulation, while `nlsr::nlxb()` proceeds as usual.

```
bdf2 <-  (x1 + t * x2 - exp(t))^2 ~ - (x3 + x4 * sin(t) - cos(t))^2

nlsbd2 <- try(nls(bdf2, start=prm0, data=bddata, trace=FALSE))
```

```
## Error in eval(formula[[2L]], data, env) : object 'x1' not found
```

```
nlsbd2
```

```
## [1] "Error in eval(formula[[2L]], data, env) : object 'x1' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in eval(formula[[2L]], data, env): object 'x1' not found>
```

```
nlsLMbd2 <- try(nlsLM(bdf2, start=prm0, data=bddata, trace=FALSE,
                      control=nls.lm.control(maxiter=10000, maxfev=10000)))
```

```
## Error in eval(formula[[2L]], data, env) : object 'x1' not found
```

```
nlsLMbd2
```

```
## [1] "Error in eval(formula[[2L]], data, env) : object 'x1' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in eval(formula[[2L]], data, env): object 'x1' not found>
```

```
nlsrbd2 <- nlxb(bdf2, start=prm0, data=bddata, trace=FALSE)
##summary(nlsrbd2)
nlsrbd2
```

```
## nlsr object: x
## residual sumsquares =  85822  on  20 observations
##     after  3056    Jacobian and  4279 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## x1              -11.5944         4.017     -2.886    0.01075     0.02637         176
## x2               13.2036         1.231      10.73  1.025e-08    -0.07207        28.1
## x3             -0.403442         28.08   -0.01437     0.9887   -0.002117       3.917
## x4              0.236777         39.79   0.005951     0.9953   -0.001648       1.624
```

We could try "prediction" again for `nlxb()`. The answers are, of course, NOT appropriate, as the `predict.nlsr()` function **ONLY** evaluates the right hand side of the formula. We are **mis-applying** the function here. It would be good to have checks on the formula to detect and warn in such cases, and I welcome collaboration to do this.

```
ndata <- data.frame(t=c(5,6), y=c(0,0))
predict(nlsrbd2, newdata=ndata)
```

```
## [1] -0.83568 -2.04425
## attr(,"class")
## [1] "predict.nlsr"
## attr(,"pkgname")
## [1] "nlsr"
```

A check on the Brown and Dennis calculation via function minimization

We can attack the Brown and Dennis problem by applying nonlinear function minimization programs to the sum of squared "residuals" as a function of the parameters. The code below does this. We omit the output for space reasons.

```
#' Brown and Dennis Function
#'
#' Test function 16 from the More', Garbow and Hillstrom paper.
#'
#' The objective function is the sum of \code{m} functions, each of \code{n}
#' parameters.
#'
#' \itemize{
#'   \item Dimensions: Number of parameters \code{n = 4}, number of summand
#'   functions \code{m >= n}.
#'   \item Minima: \code{f = 85822.2} if \code{m = 20}.
#' }
#'
#' @param m Number of summand functions in the objective function. Should be
#'   equal to or greater than 4.
#' @return A list containing:
#' \itemize{
#'   \item \code{fn} Objective function which calculates the value given input
#'   parameter vector.
#'   \item \code{gr} Gradient function which calculates the gradient vector
#'   given input parameter vector.
#'   \item \code{fg} A function which, given the parameter vector, calculates
#'   both the objective value and gradient, returning a list with members
#'   \code{fn} and \code{gr}, respectively.
#'   \item \code{x0} Standard starting point.
#' }
#' @references
#' More', J. J., Garbow, B. S., & Hillstrom, K. E. (1981).
#' Testing unconstrained optimization software.
#' \emph{ACM Transactions on Mathematical Software (TOMS)}, \emph{7}(1), 17-41.
#' \url{https://doi.org/10.1145/355934.355936}
#'
#' Brown, K. M., & Dennis, J. E. (1971).
#' \emph{New computational algorithms for minimizing a sum of squares of
#' nonlinear functions} (Report No. 71-6).
#' New Haven, CT: Department of Computer Science, Yale University.
#'
#' @examples
#' # Use 10 summand functions
#' fun <- brown_den(m = 10)
#' # Optimize using the standard starting point
#' x0 <- fun$x0
#' res_x0 <- stats::optim(par = x0, fn = fun$fn, gr = fun$gr, method =
#' "L-BFGS-B")
#' # Use your own starting point
#' res <- stats::optim(c(0.1, 0.2, 0.3, 0.4), fun$fn, fun$gr, method =
#' "L-BFGS-B")
#'
#' # Use 20 summand functions
```

```r
#' fun20 <- brown_den(m = 20)
#' res <- stats::optim(fun20$x0, fun20$fn, fun20$gr, method = "L-BFGS-B")
#' @export
#`
brown_den <- function(m = 20) {
  list(
    fn = function(par) {
      x1 <- par[1]
      x2 <- par[2]
      x3 <- par[3]
      x4 <- par[4]

      ti <- (1:m) * 0.2
      l <- x1 + ti * x2 - exp(ti)
      r <- x3 + x4 * sin(ti) - cos(ti)
      f <- l * l + r * r
      sum(f * f)
    },
    gr = function(par) {
      x1 <- par[1]
      x2 <- par[2]
      x3 <- par[3]
      x4 <- par[4]

      ti <- (1:m) * 0.2
      sinti <- sin(ti)
      l <- x1 + ti * x2 - exp(ti)
      r <- x3 + x4 * sinti - cos(ti)
      f <- l * l + r * r
      lf4 <- 4 * l * f
      rf4 <- 4 * r * f
      c(
        sum(lf4),
        sum(lf4 * ti),
        sum(rf4),
        sum(rf4 * sinti)
      )
    },
    fg = function(par) {
      x1 <- par[1]
      x2 <- par[2]
      x3 <- par[3]
      x4 <- par[4]

      ti <- (1:m) * 0.2
      sinti <- sin(ti)
      l <- x1 + ti * x2 - exp(ti)
      r <- x3 + x4 * sinti - cos(ti)
      f <- l * l + r * r
      lf4 <- 4 * l * f
      rf4 <- 4 * r * f

      fsum <- sum(f * f)
      grad <- c(
        sum(lf4),
        sum(lf4 * ti),
        sum(rf4),
        sum(rf4 * sinti)
      )

      list(
        fn = fsum,
        gr = grad
      )
    },
    x0 = c(25, 5, -5, 1)
  )
}
mbd <- brown_den(m=20)
mbd
mbd$fg(mbd$x0)
bdsolnm <- optim(mbd$x0, mbd$fn, control=list(trace=0))
bdsolnm
bdsolbfgs <- optim(mbd$x0, mbd$fn, method="BFGS", control=list(trace=0))
bdsolbfgs

library(optimx)
methlist <- c("Nelder-Mead","BFGS","Rvmmin","L-BFGS-B","Rcgmin","ucminf")

solo <- opm(mbd$x0, mbd$fn, mbd$gr, method=methlist, control=list(trace=0))
summary(solo, order=value)

## A failure above is generally because a package in the 'methlist' is not installed.
```

References

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

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

Levenberg, Kenneth. 1944. "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics* 2: 164–68.

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

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. Hilger, Bristol :

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