Jacobian Calculations for nls()

Arkajyoti Bhattacharjee, Indian Institute of Technology, Kanpur
John C. Nash, University of Ottawa, Canada

26/05/2021

Contents

ISSUES

- ExDerivs.R file causes a number of failures in the ORIGINAL numericDeriv.
- Need to verify nlsalt:: version of numericDeriv() matches all cases of nlspkg:: version
- Do we need to get a model frame? How? and How to use it?

TODOS (mostly from nlsr vignette nlsr-devdoc.Rmd)

- how to insert numerical derivatives when Deriv unable to get result (nlsr)

- approximations for jacfn beyond fwd approximation. How to specify??
- how to force numerical approximations in nlfb() in a manner consistent with that used in `optimx::optimr()`, that is, to surround the name of `jacfn` with quotes if it is a numerical approximation, or to provide a logical control to `nlxb()` for this purpose.

Jacobians in nls()

This document source is in file **DerivsNLS.Rmd**.

`nls()` and other nonlinear least squares programs in R need a Jacobian matrix calculated at the current set of trial nonlinear model parameters to set up the Gauss-Newton equations or their stabilized modifications in methods such as that of Marquardt (Marquardt (1963)). Unfortunately, `nls()` calls the Jacobian the "gradient," and uses function `numericDeriv()` to compute them. This document is an attempt to describe different ways to compute the Jacobian for use in nls() and related software, and to evaluate these approaches from several perspectives.

In evaluating performance, we need to know the conditions under which the evaluation was conducted. Thus the computations included in this document, which is built using `Rmarkdown`, are specific to the computer in which the document is processed. We will add tables that give the results for different computing environments at the bottom.

An example problem

We will use the Hobbs weed infestation problem (Nash (1979), page 120).

```r
# Data for Hobbs problem
ydat  <-  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) # for testing
tdat  <-  seq_along(ydat) # for testing

# A simple starting vector -- must have named parameters for nlxb, nls, wrapnlsr.
start1  <-  c(b1=1, b2=1, b3=1)
eunsc  <-   y ~ b1/(1+b2*exp(-b3*tt)) # formula -- display structure with str(eunsc)
# Can we convert a string form of this "model" to a formula
ceunsc <- " y ~ b1/(1+b2*exp(-b3*tt))" # This will give character form: str(ceunsc)
# Next line Will be TRUE if we have made the conversion OK
print(as.formula(ceunsc)==eunsc)
```

```
## [1] TRUE
```

```r
weeddata1  <-  data.frame(y=ydat, tt=tdat) ## LOCAL DATA IN DATA FRAMES
weedenv <- list2env(weeddata1) ## Put data in an Environment
# Add the parameter data as "variables"
weedenv$b1 <- start1[[1]]; weedenv$b2 <- start1[[2]]; weedenv$b3 <- start1[[3]]
# Display content of the Environment with ## ls.str(weedenv)
# We are now set up for computations
```

Tools for Jacobians

There are a number of ways to get the Jacobian in R.

numericDeriv() original version from base R

`numericDeriv` is the R function used by `nls()` to evaluate Jacobians for its Gauss-Newton equations. The R source code is in the file `nls.R`. It calls a C function numeric_deriv in `nls.c`. These have been extracted in an R package form as `nlspkg` by Duncan Murdoch as described in our document **PkgFromRbase.Rmd: Making a package from base R files**, and we will use that version.

In the following we will test and time `numericDeriv()` along with various of its options.

```
rexpr<-call("-",eunsc[[3]], eunsc[[2]]) # Generate the residual "call"
res0<-eval(rexpr, weedenv) # Get the residuals
print(res0) # the base residuals
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
```

```
cat("Sumsquares at 1,1,1 is ",sum(res0^2),"\n")
```

```
## Sumsquares at 1,1,1 is  23520.58
```

```
rexpr<-call("-",eunsc[[3]], eunsc[[2]]) # This is the "call" that computes the residual
## Try the numericDeriv option
theta<-names(start1)
## suppressMessages(library(nlspkg))
suppressMessages(ndnls<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv))
print(ndnls)
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
## attr(,"gradient")
##            [,1]          [,2]         [,3]
##  [1,] 0.7310585 -1.966119e-01 0.1966118813
##  [2,] 0.8807971 -1.049936e-01 0.2099871635
##  [3,] 0.9525741 -4.517674e-02 0.1355299950
##  [4,] 0.9820137 -1.766276e-02 0.0706508160
##  [5,] 0.9933071 -6.648064e-03 0.0332403183
##  [6,] 0.9975274 -2.466440e-03 0.0147991180
##  [7,] 0.9990890 -9.102821e-04 0.0063714981
##  [8,] 0.9996643 -3.356934e-04 0.0026817322
##  [9,] 0.9998765 -1.235008e-04 0.0011105537
## [10,] 0.9999547 -4.529953e-05 0.0004539490
## [11,] 0.9999828 -1.716614e-05 0.0001831055
## [12,] 0.9999943 -5.722046e-06 0.0000743866
```

```
print(sum(ndnls^2))
```

```
## [1] 23520.58
```

```
tndnls<-microbenchmark(ndnls<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv))
print(tndnls)
```

```
## Unit: microseconds
##                                                        expr   min     lq
##  ndnls <- nlspkg::numericDeriv(rexpr, theta, rho = weedenv) 10.37 10.771
##      mean median    uq    max neval
##  11.74108 11.061 11.48 66.357   100
```

```
## numericDeriv also has central difference option, as well as choice of eps parameter
## Central diff
ndnlsc<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE)
print(ndnlsc)
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
## attr(,"gradient")
##            [,1]          [,2]         [,3]
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102211e-04 6.371548e-03
##  [8,] 0.9996647 -3.352378e-04 2.681902e-03
##  [9,] 0.9998766 -1.233799e-04 1.110414e-03
## [10,] 0.9999546 -4.539623e-05 4.539581e-04
## [11,] 0.9999833 -1.670090e-05 1.837134e-04
## [12,] 0.9999939 -6.143885e-06 7.372897e-05
```

```
print(sum(ndnlsc^2))
```

```
## [1] 23520.58
```

```
tndnlsc<-microbenchmark(ndnlsc<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE))
print(tndnlsc)
```

```
## Unit: microseconds
##                                                                        expr
##  ndnlsc <- nlspkg::numericDeriv(rexpr, theta, rho = weedenv, central = TRUE)
##     min     lq     mean median      uq    max neval
##  12.957 13.257 13.97593 13.457 13.7985 53.787   100
```

```
## Forward diff with smaller eps
ndnlsx<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, eps=1e-10)
print(ndnlsx)
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
## attr(,"gradient")
##            [,1]          [,2]         [,3]
##  [1,] 0.7310597 -0.1966160568 0.1966071750
##  [2,] 0.8807977 -0.1049915710 0.2099920238
##  [3,] 0.9525714 -0.0451905180 0.1355182633
##  [4,] 0.9820056 -0.0176747506 0.0706457115
##  [5,] 0.9933032 -0.0066435746 0.0332534000
##  [6,] 0.9975309 -0.0024513724 0.0148148160
##  [7,] 0.9990941 -0.0009237056 0.0063593575
##  [8,] 0.9996626 -0.0003552714 0.0026290081
##  [9,] 0.9998757 -0.0001421085 0.0011368684
## [10,] 0.9999468  0.0000000000 0.0004973799
## [11,] 0.9998757 -0.0001421085 0.0001421085
## [12,] 1.0000178  0.0000000000 0.0001421085
```

```
print(sum(ndnlsx^2))
```

```
## [1] 23520.58
```

```
tndnlsx<-microbenchmark(ndnlsx<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, eps=1e-10))
print(tndnlsx)
```

```
## Unit: microseconds
##                                                                        expr   min
##  ndnlsx <- nlspkg::numericDeriv(rexpr, theta, rho = weedenv, eps = 1e-10) 9.197
##    lq     mean median     uq    max neval
##  9.41 10.11952 9.5535 9.8585 39.722   100
```

```
## Central diff with smaller eps
ndnlscx<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, eps=1e-10)
print(ndnlscx)
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
## attr(,"gradient")
##            [,1]          [,2]         [,3]
##  [1,] 0.7310597 -1.966116e-01 1.966116e-01
##  [2,] 0.8807977 -1.049916e-01 2.099876e-01
##  [3,] 0.9525714 -4.518164e-02 1.355271e-01
##  [4,] 0.9820145 -1.766587e-02 7.065459e-02
##  [5,] 0.9933032 -6.643575e-03 3.325340e-02
##  [6,] 0.9975309 -2.451372e-03 1.481482e-02
##  [7,] 0.9990941 -9.059420e-04 6.359357e-03
##  [8,] 0.9996981 -3.197442e-04 2.664535e-03
##  [9,] 0.9998757 -1.421085e-04 1.136868e-03
## [10,] 0.9999468 -3.552714e-05 4.618528e-04
## [11,] 0.9999468 -7.105427e-05 2.131628e-04
## [12,] 1.0000178  0.000000e+00 7.105427e-05
```

```
print(sum(ndnlscx^2))
```

```
## [1] 23520.58
```

```
tndnlscx<-microbenchmark(ndnlscx<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, eps=1e-
print(tndnlscx)
```

```
## Unit: microseconds
##                                                                                      expr
##  ndnlscx <- nlspkg::numericDeriv(rexpr, theta, rho = weedenv,      central = TRUE, eps = 1e-10)
##     min      lq     mean  median    uq    max neval
##  11.854 12.2725 13.06421 12.4415 12.71 44.404   100
```

```
## Add dir parameter -- the direction of the parameter shift
ndnlsd<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, dir=-1)
# Does dir make a difference? This might be accidental for forward difference.
max(abs(ndnlsd-ndnls))
```

```
## [1] 0
```

```
ndnlscd<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, dir=-1)
# Does dir make a difference? For central diff it should NOT!
max(abs(ndnlscd-ndnlsc))
```

```
## [1] 0
```

numericDeriv() alternative pure-R version

This version (see Appendix 2) has C code replaced with R equivalents.

```
## Try ExDerivs.R ??
suppressMessages(andnls<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv))
# print(andnls); print(sum(andnls^2))
```

```r
tandnls<-microbenchmark(andnls<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv))
print(tandnls)
```

```
## Unit: microseconds
##                                                         expr   min     lq
##  andnls <- nlsalt::numericDeriv(rexpr, theta, rho = weedenv) 30.66 31.659
##      mean  median      uq    max neval
##  35.76053 32.3275 33.3205 82.424   100
```

```r
## numericDeriv also has central difference option, as well as choice of eps parameter
## Central diff
andnlsc<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE)
ndnlsc<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE)
# print(andnlsc); print(sum(andnlsc^2))
tandnlsc<-microbenchmark(andnlsc<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE))
print(tandnlsc)
```

```
## Unit: microseconds
##                                                                             expr
##  andnlsc <- nlsalt::numericDeriv(rexpr, theta, rho = weedenv,      central = TRUE)
##     min      lq     mean  median      uq   max neval
##  40.792 41.5425 43.47624 41.9825 42.7615 89.09   100
```

```r
## Forward diff with smaller eps
andnlsx<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, eps=1e-10)
ndnlsx<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, eps=1e-10)
# print(andnlsx); print(sum(andnlsx^2))
tandnlsx<-microbenchmark(andnlsx<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, eps=1e-10))
print(tandnlsx)
```

```
## Unit: microseconds
##                                                                          expr
##  andnlsx <- nlsalt::numericDeriv(rexpr, theta, rho = weedenv,      eps = 1e-10)
##     min     lq     mean median      uq    max neval
##  29.715 30.404 31.51655 30.785 31.1455 75.795   100
```

```r
## Central diff with smaller eps
andnlscx<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, eps=1e-10)
ndnlscx<-nlspkg::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, eps=1e-10)
# print(andnlscx) ; print(sum(andnlscx^2))
tandnlscx<-microbenchmark(andnlscx<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, eps=1e
print(tandnlscx)
```

```
## Unit: microseconds
##                                                                                           expr
##  andnlscx <- nlsalt::numericDeriv(rexpr, theta, rho = weedenv,      central = TRUE, eps = 1e-10)
##     min      lq     mean median      uq     max neval
##  40.005 41.0585 43.11109  41.58 42.5515 118.096   100
```

```r
## Comparisons for Jacobian between nlspkg and nlsalt i.e. R&C vs just R
max(abs(attr(ndnls, "gradient")-attr(andnls,"gradient")))
```

```
## [1] 0
```

```r
max(abs(attr(ndnlsc, "gradient")-attr(andnlsc,"gradient")))
```

```
## [1] 0
```

```
max(abs(attr(ndnlsx, "gradient")-attr(andnlsx,"gradient")))
```

```
## [1] 0
```

```
max(abs(attr(ndnlscx, "gradient")-attr(andnlscx,"gradient")))
```

```
## [1] 0
```

```
## Using dir
cat("eps (regular) = ",.Machine$double.eps^(1/2),
    "  eps (central) =",.Machine$double.eps^(1/3),"\n")
```

```
## eps (regular) =  1.490116e-08   eps (central) = 6.055454e-06
```

```
andnlsd<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, dir=-1)
max(abs(attr(andnlsd, "gradient")-attr(ndnls,"gradient")))
```

```
## [1] 9.536743e-07
```

```
max(abs(attr(andnlsd, "gradient")-attr(andnls,"gradient")))
```

```
## [1] 9.536743e-07
```

```
andnlscd<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, dir=-1)
max(abs(attr(andnlscd, "gradient")-attr(ndnlsc,"gradient")))
```

```
## [1] 0
```

```
## Try comparisons over different eps sizes
for (ee in 3:10){
andnlsd<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, dir=-1, eps=10^(-ee))
andnlscd<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, dir=-1, eps=10^(-ee))
andnlsx<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, eps=10^(-ee))
andnlscx<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv, central=TRUE, eps=10^(-ee))
cat("Regular diff, eps=10^(-",ee,"):",
      max(abs(attr(andnlsd,"gradient")-attr(andnlsx,"gradient"))),"\n")
cat("Central diff, eps=10^(-",ee,"):",
    max(abs(attr(andnlscd,"gradient")-attr(andnlscx,"gradient"))),"\n")
}
```

```
## Regular diff, eps=10^(- 3 ): 0.0003680243
## Central diff, eps=10^(- 3 ): 0
## Regular diff, eps=10^(- 4 ): 3.680242e-05
## Central diff, eps=10^(- 4 ): 0
## Regular diff, eps=10^(- 5 ): 3.680256e-06
## Central diff, eps=10^(- 5 ): 0
## Regular diff, eps=10^(- 6 ): 3.677059e-07
## Central diff, eps=10^(- 6 ): 0
## Regular diff, eps=10^(- 7 ): 1.421085e-07
## Central diff, eps=10^(- 7 ): 0
## Regular diff, eps=10^(- 8 ): 1.421085e-06
## Central diff, eps=10^(- 8 ): 0
## Regular diff, eps=10^(- 9 ): 1.421085e-05
## Central diff, eps=10^(- 9 ): 0
## Regular diff, eps=10^(- 10 ): 0.0001421085
## Central diff, eps=10^(- 10 ): 0
```

The `dir` parameter allows us to use a backward difference for the derivative. This appears in `nlsModel()` for the case where a parameter is on an upper bound for the case `algorithm="port"`. It does not check for nearness to the bound, and for the lower bound assumes that we are stepping AWAY from the bound in the default direction (`dir=+1`). None of the code addresses the issue where bounds are closer together than the step used for the finite difference, so there are situations where we could crash the code. Nor does the code check if the central difference is specified when near a bound.

- In the case of lower bounds, a central difference can overstep the bound when a parameter is "close" or on the bound.
- In the case of an upper bound, changing the `dir` will not change the central derivative approximation expression and steps in both forward and backward directions of the parameter are taken.

Symbolic methods from `nlsr`

The package `nlsr` has a function `model2rjfun()` that converts an expression describing how the residual functions are computed into an R function that computes the residuals at a particular set of parameters and sets the **attribute** "gradient" of the vector of residual values to the Jacobian at the particular set of parameters.

```
# nlsr has function model2rjfun. We can evaluate just the residuals
res0<-model2rjfun(eunsc, start1, data=weeddata1, jacobian=FALSE)
res0(start1)
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
```

```
# or the residuals and jacobian
## nlsr::model2rjfun forms a function with gradient (jacobian) attribute
funsc <- model2rjfun(eunsc, start1, data=weeddata1) # from nlsr: creates a function
tmodel2rjfun <- microbenchmark(model2rjfun(eunsc, start1, data=weeddata1))
print(tmodel2rjfun)
```

```
## Unit: microseconds
##                                          expr    min     lq     mean median
##  model2rjfun(eunsc, start1, data = weeddata1) 81.553 82.785 87.26409 83.616
##      uq     max neval
##  85.263 213.224   100
```

```
print(funsc)
```

```
## function(prm) {
##         if (is.null(names(prm)))
##      names(prm) <- names(pvec)
##    localdata <- list2env(as.list(prm), parent = data)
##    eval(residexpr, envir = localdata)
##         # Saves Jacobian matrix as "gradient" attribute (consistent with deriv())
##     }
## <bytecode: 0x55d5ea5dcae8>
## <environment: 0x55d5f107c1c0>
```

```
print(funsc(start1))
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
## attr(,"gradient")
##             b1            b2           b3
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102212e-04 6.371548e-03
##  [8,] 0.9996646 -3.352377e-04 2.681901e-03
##  [9,] 0.9998766 -1.233793e-04 1.110414e-03
## [10,] 0.9999546 -4.539581e-05 4.539581e-04
## [11,] 0.9999833 -1.670114e-05 1.837126e-04
## [12,] 0.9999939 -6.144137e-06 7.372964e-05
```

```
print(environment(funsc))
```

```
## <environment: 0x55d5f107c1c0>
```

```
print(ls.str(environment(funsc)))
```

```
## data : <environment: 0x55d5f128f210>
## jacobian :  logi TRUE
## modelformula : Class 'formula'  language y ~ b1/(1 + b2 * exp(-b3 * tt))
## pvec :  Named num [1:3] 1 1 1
## residexpr :   expression({  .expr3 <- exp(-b3 * tt)  .expr5 <- 1 + b2 * .expr3  .expr10 <- .expr5^2
## rjfun : function (prm)
## testresult :  logi TRUE
```

```
print(ls(environment(funsc)$data))
```

```
## [1] "tt" "y"
```

```
eval(eunsc, environment(funsc))
```

```
## y ~ b1/(1 + b2 * exp(-b3 * tt))
```

```
vfunsc<-funsc(start1)
print(vfunsc)
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
## attr(,"gradient")
##              b1            b2           b3
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102212e-04 6.371548e-03
##  [8,] 0.9996646 -3.352377e-04 2.681901e-03
##  [9,] 0.9998766 -1.233793e-04 1.110414e-03
## [10,] 0.9999546 -4.539581e-05 4.539581e-04
## [11,] 0.9999833 -1.670114e-05 1.837126e-04
## [12,] 0.9999939 -6.144137e-06 7.372964e-05
```

```
tfunsc<-microbenchmark(funsc(start1))
print(tfunsc)
```

```
## Unit: microseconds
##           expr   min     lq     mean  median     uq    max neval
##  funsc(start1) 15.35 16.699 17.63718 16.8945 17.231 48.403   100
```

`numDeriv` package

The package `numDeriv` includes a function `jacobian()` that acts on a user function `resid()` to produce the Jacobian at a set of parameters by several choices of approximation.

```
# We use the residual function (without gradient attribute) from nlsr
jnlsc<-jacobian(res0, start1)
jnlsc
```

```
##            [,1]          [,2]         [,3]
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102212e-04 6.371548e-03
##  [8,] 0.9996647 -3.352378e-04 2.681902e-03
##  [9,] 0.9998766 -1.233791e-04 1.110414e-03
## [10,] 0.9999546 -4.539572e-05 4.539580e-04
## [11,] 0.9999833 -1.670116e-05 1.837129e-04
## [12,] 0.9999939 -6.144205e-06 7.373002e-05
```

```
# Timings of the analytic jacobian calculations
tjnlsc<-microbenchmark(jnlsc<-jacobian(res0, start1))
print(tjnlsc)
```

```
## Unit: microseconds
##                            expr     min       lq     mean   median      uq
##  jnlsc <- jacobian(res0, start1) 344.413 351.9415 360.6021 355.1935 362.016
##      max neval
##  497.917   100
```

Note that the manual pages for `numDeriv` offer many options for the functions in the package. At 2021-5-27 we have yet to explore these.

Comparisons

In the following, we are comparing to `vfunsc`, which is the evaluated residual vector at `start1=c(1,1,1)` with "gradient" attribute (jacobian) included, as developed using package `nlsr`. This is taken as the "correct" result, even though it is possible that the generated order of calculations may introduce inaccuracies in the supposedly analytic derivatives.

`numericDeriv` computes a similar structure (residuals with "gradient" attribute): `ndnlsc`: the forward difference result with default `eps` (1e-07 according to manual) `ndnlsc2`: Central difference with default `eps` `ndnlscx`: Forward difference with smaller eps=1e-10 `ndnlscx2`: Central difference with smaller eps=1e-10

`jnlsc`: numDeriv::jacobian() result with default settings.

```
## Matrix comparisons
attr(ndnls, "gradient")-attr(vfunsc,"gradient")
```

```
##                 b1            b2            b3
##  [1,] -4.066995e-08 -7.619266e-09 -5.198538e-08
##  [2,]  1.016833e-08  3.631656e-09 -7.263312e-09
##  [3,]  7.050552e-09 -8.473015e-08  1.577186e-08
##  [4,] -8.764533e-08 -5.738229e-08 -8.889419e-09
##  [5,] -3.542825e-08 -6.988878e-09  3.494439e-08
##  [6,] -1.592723e-08  6.909055e-08  6.229383e-08
##  [7,]  5.380365e-08 -6.095489e-08 -5.015294e-08
##  [8,] -3.432289e-07 -4.556886e-07 -1.691883e-07
##  [9,] -1.062480e-07 -1.214742e-07  1.395936e-07
## [10,]  9.833867e-08  9.627771e-08 -9.102750e-09
## [11,] -4.647158e-07 -4.649948e-07 -6.071033e-07
## [12,]  4.221287e-07  4.220910e-07  6.569545e-07
```

```
attr(ndnlsc, "gradient")-attr(vfunsc,"gradient")
```

```
##                  b1            b2            b3
##  [1,] -5.513268e-11  1.371020e-11  5.962686e-11
##  [2,]  6.850076e-13 -3.831403e-11  3.291006e-12
##  [3,] -2.144829e-11 -1.208666e-10  6.925160e-11
##  [4,] -2.665634e-11  4.123477e-11 -1.826495e-11
##  [5,]  2.175706e-10 -7.825720e-11  9.793778e-11
##  [6,] -2.416068e-10 -1.666460e-10 -1.735172e-10
##  [7,] -8.350343e-11  5.415257e-11 -8.571976e-11
##  [8,]  3.255913e-10 -9.864836e-11  2.024904e-10
##  [9,]  1.379652e-11 -5.685668e-10 -1.631673e-10
## [10,] -9.296786e-11 -4.173983e-10  6.710748e-11
## [11,] -4.266865e-11  2.415372e-10  8.632699e-10
## [12,] -2.738492e-10  2.515432e-10 -6.717320e-10
```

```
attr(ndnlsx, "gradient")-attr(vfunsc,"gradient")
```

```
##                  b1            b2            b3
##  [1,]  1.078625e-06 -4.123528e-06 -4.758257e-06
##  [2,]  5.790545e-07  2.014411e-06  4.852963e-06
##  [3,] -2.771694e-06 -1.385826e-05 -1.171591e-05
##  [4,] -8.202081e-06 -1.204434e-05 -5.113350e-06
##  [5,] -3.931620e-06  4.482091e-06  1.311668e-05
##  [6,]  3.569890e-06  1.513685e-05  1.576029e-05
##  [7,]  5.191947e-06 -1.348438e-05 -1.219078e-05
##  [8,] -2.074929e-06 -2.003370e-05 -5.289324e-05
##  [9,] -8.676624e-07 -1.872920e-05  2.645423e-05
## [10,] -7.810096e-06  4.539581e-05  4.342184e-05
## [11,] -1.075608e-04 -1.254074e-04 -4.160402e-05
## [12,]  2.399048e-05  6.144137e-06  6.837890e-05
```

```
attr(ndnlscx, "gradient")-attr(vfunsc,"gradient")
```

```
##                  b1            b2            b3
##  [1,]  1.078625e-06  3.173646e-07 -3.173646e-07
##  [2,]  5.790545e-07  2.014411e-06  4.120706e-07
##  [3,] -2.771694e-06 -4.976479e-06 -2.834131e-06
##  [4,]  6.797035e-07 -3.162555e-06  3.768434e-06
##  [5,] -3.931620e-06  4.482091e-06  1.311668e-05
##  [6,]  3.569890e-06  1.513685e-05  1.576029e-05
##  [7,]  5.191947e-06  4.279192e-06 -1.219078e-05
##  [8,]  3.345221e-05  1.549344e-05 -1.736611e-05
##  [9,] -8.676624e-07 -1.872920e-05  2.645423e-05
## [10,] -7.810096e-06  9.868671e-06  7.894701e-06
## [11,] -3.650654e-05 -5.435313e-05  2.945025e-05
## [12,]  2.399048e-05  6.144137e-06 -2.675369e-06
```

```
jnlsc-attr(vfunsc,"gradient")
```

```
##                  b1            b2            b3
##  [1,] -2.239464e-11  7.806283e-12 -1.156686e-12
##  [2,] -2.267631e-12  2.974312e-11  2.957756e-11
##  [3,] -8.948509e-12  3.630193e-11 -1.256267e-11
##  [4,] -1.649125e-12  1.182179e-13  6.369841e-11
##  [5,] -4.272493e-11 -1.109757e-11  3.501116e-11
##  [6,]  1.867381e-10  1.793287e-11  3.319552e-11
##  [7,]  1.090728e-11  1.840947e-11  9.946462e-12
##  [8,]  2.035664e-10 -1.520996e-10  1.911593e-10
##  [9,] -3.582228e-10  2.039028e-10 -1.905254e-10
## [10,]  3.202474e-10  8.291927e-11 -6.263366e-11
## [11,]  5.931922e-12 -1.779933e-11  3.682277e-10
## [12,]  4.132902e-10 -6.831839e-11  3.817154e-10
```

```
## Summary comparisons
max(abs(attr(ndnls, "gradient")-attr(vfunsc,"gradient")))
```

```
## [1] 6.569545e-07
```

```
max(abs(attr(ndnlsc, "gradient")-attr(vfunsc,"gradient")))
```

```
## [1] 8.632699e-10
```

```
max(abs(attr(ndnlsx, "gradient")-attr(vfunsc,"gradient")))
```

```
## [1] 0.0001254074
```

```
max(abs(attr(ndnlscx, "gradient")-attr(vfunsc,"gradient")))
```

```
## [1] 5.435313e-05
```

```
max(abs(jnlsc-attr(vfunsc,"gradient")))
```

```
## [1] 4.132902e-10
```

Performance results for different computing environments

Here we present tables of the results, preceded by identified descriptions of the machines we used. We use ideas and functions from the document `MachineSummary` to provide a characterization and identity for each machine used.

M21-LM20.1

?? still to be run

?? What machines provide a range of possibilities.

Discussion of derivative computation for nonlinear least squares

In no particular order, we comment on some issues relating to the Jacobian calculations in nonlinear least squares.

Nomenclature

R is not in step with many other areas of numerical computation when labelling different objects in the nonlinear least squares problem. In particular, R uses the term "gradient" when the object of interest is the Jacobian matrix. In that it is useful in performing iterations of the Gauss-Newton or related equations to have the Jacobian associated with the residuals, and the rows of the Jacobian matrix are "gradients" of the respective residuals, we can accept the attribute name "gradient" to select the required information. Moreover, as in package `nlsr` it is very useful to have the Jacobian matrix as an attribute of the residual vector, since the main solver function, in this case `nlsr::nlfb()`, can be called with the same input for the arguments `res` and `jac`. These are the functions required to compute the residual and the Jacobian, and using the same function for both is very convenient, but needs some way to return both the residual vector and Jacobian matrix in a coherent fashion.

Numerical approximation near constraints

As far as we are aware there is no software that implements a fully safeguarded system to compute numerical approximation of the Jacobian (or gradients in general optimization) near constraints. The same statement applies even in the case of the much simpler bounds constraints. Users have a perverse tendency to devise ways to foil our best efforts. For example, they may decide that a good way to specify fixed (i.e., masked) parameters that they do not want to vary during a particular calculation is to specify the lower and upper bound of a parameter at the same value. Later runs may want the parameter constraints relaxed.

In `nlsr::nlxb()`, users may, in fact, specify masked parameters this way. This is a case of "if you can't beat them, join them," but it does provide an easily understood way for users to fix values.

More tricky is dealing with constraints that are close together. Note that these may arise from, for example, two linear (planar) constraints that approach at a narrow angle. In the apex where these constraints intersect, we will have tight bounds on parameters. If the constraint is not one that is imposed by the nature of the residual or objective function, for example, a log() or square root near zero, then we can generally proceed and allow the derivative approximator to evaluate outside the constraints. Things are decidedly nastier if we do have inadmissible values.

The issue of constraints and the need for a step in parameter values for derivative approximations was one of the motivations for trying to find analytic derivatives in package `nlsr` and the continuing effort to bring them into other R tools.

A case where the initial Jacobian is singular

The following example shows that numericDeriv() does a reasonable job of computing the Jacobian, but the result is still singular.

```r
# File: badJlogmod.R
# A problem illustrating poor numeric Jacobian
form<-y ~ 10*a*(8*b-log(0.075*c*x)) # the model formula
# This model uses log near a small argument, which skirts the dangerous
# value of 0. The parameters a, b, c could all be 1 "safely" as a start.
x<-1:20 # define x
a<-1.01
b<-.9
c<-.95
y <- 10*a*(8*b-log(0.075*c*x))+0.2*runif(20) # compute a y
df<-data.frame(x=x, y=y)
# plot(x,y) # for information
st<-c(a=1, b=1,c=1) # set the "default" starting vector
n0<-try(nls(form, start=st, data=df)) # and watch the fun as this fails.
```

```
## Error in nlsModel(formula, mf, start, wts, scaleOffset = scOff, nDcentral = nDcntr) :
##   singular gradient matrix at initial parameter estimates
```

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

```
n1
```

```
## nlsr object: x
## residual sumsquares =  0.065093  on  20 observations
##     after  5    Jacobian and  6 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                1.00967           NA         NA         NA  -5.161e-08       501.9
## b               0.950581           NA         NA         NA   2.775e-08        25.6
## c                1.40495           NA         NA         NA  -2.469e-09   6.371e-15
```

```
jmod<-model2rjfun(form, pvec=st,data=data.frame(x=x, y=y)) # extract the model
Jatst<-jmod(st) # compute this at the start from package nlsr
Jatst<-attr(Jatst,"gradient") # and extract the Jacobian
#
# Now try to compute Jacobian produced by nls()
env<-environment(form) # We need the environment of the formula
eform<-eval(form, envir=env) # and the evaluated expression
localdata<-list2env(as.list(st), parent=env)
jnlsatst<-numericDeriv(form[[3L]], theta=names(st), rho=localdata)
Jnls<-attr(jnlsatst,"gradient")
Jnls # from nls()
```

```
##          [,1] [,2] [,3]
##  [1,] 105.903   80  -10
##  [2,]  98.971   80  -10
##  [3,]  94.917   80  -10
##  [4,]  92.040   80  -10
##  [5,]  89.808   80  -10
##  [6,]  87.985   80  -10
##  [7,]  86.444   80  -10
##  [8,]  85.108   80  -10
##  [9,]  83.930   80  -10
## [10,]  82.877   80  -10
## [11,]  81.924   80  -10
## [12,]  81.054   80  -10
## [13,]  80.253   80  -10
## [14,]  79.512   80  -10
## [15,]  78.822   80  -10
## [16,]  78.177   80  -10
## [17,]  77.571   80  -10
## [18,]  76.999   80  -10
## [19,]  76.458   80  -10
## [20,]  75.945   80  -10
```

```
Jatst # from nlsr -- analytic derivative
```

```
##             a  b   c
##  [1,] 105.903 80 -10
##  [2,]  98.971 80 -10
##  [3,]  94.917 80 -10
##  [4,]  92.040 80 -10
##  [5,]  89.808 80 -10
##  [6,]  87.985 80 -10
##  [7,]  86.444 80 -10
##  [8,]  85.108 80 -10
##  [9,]  83.930 80 -10
## [10,]  82.877 80 -10
## [11,]  81.924 80 -10
## [12,]  81.054 80 -10
## [13,]  80.253 80 -10
## [14,]  79.512 80 -10
## [15,]  78.822 80 -10
## [16,]  78.177 80 -10
## [17,]  77.571 80 -10
## [18,]  76.999 80 -10
## [19,]  76.458 80 -10
## [20,]  75.945 80 -10
```

```
max(abs(Jnls-Jatst))
```

```
## [1] 9.5367e-07
```

```
svd(Jnls)$d
```

```
## [1] 5.2370e+02 2.4390e+01 9.0872e-07
```

```
svd(Jatst)$d
```

```
## [1] 5.2370e+02 2.4390e+01 1.3047e-15
```

```
# Even start at the solution?
n0c<-try(nls(form, start=coef(n1), data=data.frame(x=x, y=y), trace=TRUE))
```

```
## Error in nlsModel(formula, mf, start, wts, scaleOffset = scOff, nDcentral = nDcntr) :
##   singular gradient matrix at initial parameter estimates
```

```
## attempts with nlsj
library(nlsj)
```

```
##
## Attaching package: 'nlsj'

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

```
n0jn<-try(nlsj(form, start=st, data=df, trace=TRUE,control=nlsj.control(derivmeth="numericDeriv")))
```

```
## Warning in nlsj(form, start = st, data = df, trace = TRUE, control =
## nlsj.control(derivmeth = "numericDeriv")): Forcing numericDeriv

## control:$maxiter
## [1] 500
##
## $tol
## [1] 1e-05
##
## $minFactor
## [1] 0.00097656
##
## $printEval
## [1] FALSE
##
## $warnOnly
## [1] FALSE
```

```
##
## $scaleOffset
## [1] 0
##
## $nDcentral
## [1] FALSE
##
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 0
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 0.4
##
## $resmax
## [1] 10000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## $derivmeth
## [1] "numericDeriv"
##
## $altderivmeth
## [1] "numericDeriv"
##
## $trace
## [1] FALSE
##
## cont.
## nlsj: Using default algorithm
## maskidx:integer(0)
## lhs has just the variable  y
## npar= 3  pnames:[1] "a" "b" "c"
## Top - slam= 0  ssmin= 872.81  at [1] 1 1 1
## npar= 3
## [1] 1 1 1
## [1] 1 1 1
## i= 1  bmi= 1
## i= 2  bmi= 1
## i= 3  bmi= 1
##           [,1] [,2] [,3]
##  [1,] 105.903   80  -10
##  [2,]  98.971   80  -10
##  [3,]  94.917   80  -10
##  [4,]  92.040   80  -10
##  [5,]  89.808   80  -10
##  [6,]  87.985   80  -10
##  [7,]  86.444   80  -10
##  [8,]  85.108   80  -10
##  [9,]  83.930   80  -10
## [10,]  82.877   80  -10
## [11,]  81.924   80  -10
## [12,]  81.054   80  -10
## [13,]  80.253   80  -10
## [14,]  79.512   80  -10
## [15,]  78.822   80  -10
## [16,]  78.177   80  -10
## [17,]  77.571   80  -10
## [18,]  76.999   80  -10
## [19,]  76.458   80  -10
## [20,]  75.945   80  -10
## Error in nlsj(form, start = st, data = df, trace = TRUE, control = nlsj.control(derivmeth = "numeric
##   Singular jacobian
```

```
tmp<-readline("more.")
```

```
## more.
```

```
n0ja<-try(nlsj(form, start=st, data=df, trace=TRUE,control=nlsj.control(derivmeth="default")))
```

```
## control:$maxiter
## [1] 500
##
## $tol
## [1] 1e-05
##
## $minFactor
## [1] 0.00097656
##
## $printEval
## [1] FALSE
##
## $warnOnly
## [1] FALSE
##
## $scaleOffset
## [1] 0
##
## $nDcentral
## [1] FALSE
##
## $watch
## [1] FALSE
##
```

```
## $phi
## [1] 1
##
## $lamda
## [1] 0
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 0.4
##
## $resmax
## [1] 10000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## $derivmeth
## [1] "default"
##
## $altderivmeth
## [1] "numericDeriv"
##
## $trace
## [1] FALSE
##
## cont.
## nlsj: Using default algorithm
## maskidx:integer(0)
## lhs has just the variable  y
## npar= 3  pnames:[1] "a" "b" "c"
## Top - slam= 0  ssmin= 872.81  at [1] 1 1 1
## npar= 3
## [1] 1 1 1
## [1] 1 1 1
## i= 1  bmi= 1
## i= 2  bmi= 1
## i= 3  bmi= 1
##             a  b   c
##  [1,] 105.903 80 -10
##  [2,]  98.971 80 -10
##  [3,]  94.917 80 -10
##  [4,]  92.040 80 -10
##  [5,]  89.808 80 -10
##  [6,]  87.985 80 -10
##  [7,]  86.444 80 -10
##  [8,]  85.108 80 -10
##  [9,]  83.930 80 -10
## [10,]  82.877 80 -10
## [11,]  81.924 80 -10
## [12,]  81.054 80 -10
## [13,]  80.253 80 -10
## [14,]  79.512 80 -10
## [15,]  78.822 80 -10
## [16,]  78.177 80 -10
## [17,]  77.571 80 -10
## [18,]  76.999 80 -10
## [19,]  76.458 80 -10
## [20,]  75.945 80 -10
## Error in nlsj(form, start = st, data = df, trace = TRUE, control = nlsj.control(derivmeth = "default
##   Singular jacobian
```

Appendix 1: Base R numericDeriv code

This code is in two files, nls.R and nls.c and is extracted here.

From nls.R

```
numericDeriv <- function(expr, theta, rho = parent.frame(), dir = 1,
                 eps = .Machine$double.eps ^ (1/if(central) 3 else 2), central = FALSE)
## Note: this expr must be set up as a call to work properly according to JN??
## ?? we set eps conditional on central. But central set AFTER eps. Is this OK.
{    cat("numericDeriv-Alt\n")
    dir <- rep_len(dir, length(theta))
    stopifnot(is.finite(eps), eps > 0)
    rho1 <- new.env(FALSE, rho, 0)
    if (!is.character(theta) ) {stop("'theta' should be of type character")}
    if (is.null(rho)) {
            stop("use of NULL environment is defunct")
            #        rho <- R_BaseEnv;
    } else {
          if(! is.environment(rho)) {stop("'rho' should be an environment")}
          #    int nprot = 3;
    }
    if( ! ((length(dir) == length(theta) ) & (is.numeric(dir) ) ) )
              {stop("'dir' is not a numeric vector of the correct length") }
    if(is.na(central)) { stop("'central' is NA, but must be TRUE or FALSE") }
    res0 <- eval(expr, rho) # the base residuals. ?? C has a check for REAL ANS=res0
    if (any(is.infinite(res0)) ) {stop("residuals cannot be evaluated at base point")}
    ##  CHECK_FN_VAL(res, ans);  ?? how to do this. Is it necessary?
    nt <- length(theta) # number of parameters
    mr <- length(res0) # number of residuals
    JJ <- matrix(NA, nrow=mr, ncol=nt) # Initialize the Jacobian
    for (j in 1:nt){
       origPar<-get(theta[j],rho)
       xx <- abs(origPar)
       delta <- if (xx == 0.0) {eps} else { xx*eps }
       ## JN: I prefer eps*(xx + eps)  which is simpler ?? Should we suggest / use a control switch
       prmx<-origPar+delta*dir[j]
       assign(theta[j],prmx,rho)
       res1 <- eval(expr, rho) # new residuals (forward step)
```

```
        if (central) { # compute backward step resids for central diff
           prmb <- origPar - dir[j]*delta
           assign(theta[j], prmb, envir=rho) # may be able to make more efficient later??
           resb <- eval(expr, rho)
           JJ[, j] <- dir[j]*(res1-resb)/(2*delta) # vectorized
        } else { ## forward diff
           JJ[,j] <- dir[j]*(res1-res0)/delta
        }  # end forward diff
    } # end loop over the parameters
    attr(res0, "gradient") <- JJ
    return(res0)
}
```

From nls.c

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include <R.h>
#include <Rinternals.h>
#include "nls.h"
#include "internals.h"

#ifndef MIN
#define MIN(a,b) (((a)<(b))?(a):(b))
#endif

/*
 *  call to numeric_deriv from R -
 *  .Call("numeric_deriv", expr, theta, rho, dir = 1., eps = .Machine$double.eps, central=FALSE)
 *  Returns: ans
 */
SEXP
numeric_deriv(SEXP expr, SEXP theta, SEXP rho, SEXP dir, SEXP eps_, SEXP centr,
              SEXP rho1)
{
    if(!isString(theta))
    error(_("'theta' should be of type character"));
    if (isNull(rho)) {
    error(_("use of NULL environment is defunct"));
    rho = R_BaseEnv;
    } else
    if(!isEnvironment(rho))
        error(_("'rho' should be an environment"));
    int nprot = 3;
    if(TYPEOF(dir) != REALSXP) {
    PROTECT(dir = coerceVector(dir, REALSXP)); nprot++;
    }
    if(LENGTH(dir) != LENGTH(theta))
    error(_("'dir' is not a numeric vector of the correct length"));
    Rboolean central = asLogical(centr);
    if(central == NA_LOGICAL)
    error(_("'central' is NA, but must be TRUE or FALSE"));
```

```
//    SEXP rho1 = PROTECT(R_NewEnv(rho, FALSE, 0));
//    nprot++;
    SEXP
    pars = PROTECT(allocVector(VECSXP, LENGTH(theta))),
        ans  = PROTECT(duplicate(eval(expr, rho1)));
    double *rDir = REAL(dir),  *res = NULL; // -Wall
#define CHECK_FN_VAL(_r_, _ANS_) do {                   \
    if(!isReal(_ANS_)) {                            \
    SEXP temp = coerceVector(_ANS_, REALSXP);           \
    UNPROTECT(1);/*: _ANS_ *must* have been the last PROTECT() ! */ \
    PROTECT(_ANS_ = temp);                          \
    }                                       \
    _r_ = REAL(_ANS_);                              \
    for(int i = 0; i < LENGTH(_ANS_); i++) {                \
    if (!R_FINITE(_r_[i]))                          \
        error(_("Missing value or an infinity produced when evaluating the model")); \
    }                                       \
} while(0)

    CHECK_FN_VAL(res, ans);

    const void *vmax = vmaxget();
    int lengthTheta = 0;
    for(int i = 0; i < LENGTH(theta); i++) {
    const char *name = translateChar(STRING_ELT(theta, i));
    SEXP s_name = install(name);
    SEXP temp = findVar(s_name, rho1);
    if(isInteger(temp))
        error(_("variable '%s' is integer, not numeric"), name);
    if(!isReal(temp))
        error(_("variable '%s' is not numeric"), name);
    // We'll be modifying the variable, so need to make a copy PR#15849
    defineVar(s_name, temp = duplicate(temp), rho1);
    MARK_NOT_MUTABLE(temp);
    SET_VECTOR_ELT(pars, i, temp);
    lengthTheta += LENGTH(VECTOR_ELT(pars, i));
    }
    vmaxset(vmax);
    SEXP gradient = PROTECT(allocMatrix(REALSXP, LENGTH(ans), lengthTheta));
    double *grad = REAL(gradient);
    double eps = asReal(eps_); // was hardcoded sqrt(DOUBLE_EPS) { ~= 1.49e-08, typically}
    for(int start = 0, i = 0; i < LENGTH(theta); i++) {
    double *pars_i = REAL(VECTOR_ELT(pars, i));
    for(int j = 0; j < LENGTH(VECTOR_ELT(pars, i)); j++, start += LENGTH(ans)) {
        double
        origPar = pars_i[j],
        xx = fabs(origPar),
        delta = (xx == 0) ? eps : xx*eps;
        pars_i[j] += rDir[i] * delta;
        SEXP ans_del = PROTECT(eval(expr, rho1));
        double *rDel = NULL;
        CHECK_FN_VAL(rDel, ans_del);
        if(central) {
        pars_i[j] = origPar - rDir[i] * delta;
        SEXP ans_de2 = PROTECT(eval(expr, rho1));
        double *rD2 = NULL;
        CHECK_FN_VAL(rD2, ans_de2);
        for(int k = 0; k < LENGTH(ans); k++) {
            grad[start + k] = rDir[i] * (rDel[k] - rD2[k])/(2 * delta);
        }
        } else { // forward difference  (previously hardwired):
        for(int k = 0; k < LENGTH(ans); k++) {
            grad[start + k] = rDir[i] * (rDel[k] - res[k])/delta;
        }
        }
        UNPROTECT(central ? 2 : 1); // ansDel & possibly ans
        pars_i[j] = origPar;
    }
    }
    setAttrib(ans, install("gradient"), gradient);
    UNPROTECT(nprot);
    return ans;
}
```

Appendix 2: numericDeriv() from nlsalt package (all in R)

```
#  File src/library/stats/R/nlsnd.R
#  Part of the modified R package, https://www.R-project.org
#
#  Copyright (C) 2000-2020 The R Core Team
#  Copyright (C) 1999-1999 Saikat DebRoy, Douglas M. Bates, Jose C. Pinheiro
#  J C Nash 2021
#
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 2 of the License, or
#  (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  A copy of the GNU General Public License is available at
#  https://www.R-project.org/Licenses/

###
###            numeric Jacobian for Nonlinear least squares for R
###

numericDeriv <- function(expr, theta, rho = parent.frame(), dir = 1,
                 eps = .Machine$double.eps ^ (1/if(central) 3 else 2), central = FALSE)
## Note: this expr must be set up as a call to work properly according to JN??
## ?? we set eps conditional on central. But central set AFTER eps. Is this OK.
{   ndtrace<-FALSE
#    if(ndtrace) cat("numericDeriv-Alt\n")
    dir <- rep_len(dir, length(theta))
    stopifnot(is.finite(eps), eps > 0)
    rho1 <- new.env(FALSE, rho, 0)
    if (!is.character(theta) ) {stop("'theta' should be of type character")}
    if (is.null(rho)) {
            stop("use of NULL environment is defunct")
            #        rho <- R_BaseEnv;
    } else {
          if(! is.environment(rho)) {stop("'rho' should be an environment")}
          #    int nprot = 3;
    }
    if( ! ((length(dir) == length(theta) ) & (is.numeric(dir) ) ) )
              {stop("'dir' is not a numeric vector of the correct length") }
    if(is.na(central)) { stop("'central' is NA, but must be TRUE or FALSE") }
    res0 <- eval(expr, rho) # the base residuals. ?? C has a check for REAL ANS=res0
    if (any(is.infinite(res0)) ) {stop("residuals cannot be evaluated at base point")}
    ##  CHECK_FN_VAL(res, ans);  ?? how to do this. Is it necessary?
    nt <- length(theta) # number of parameters
    mr <- length(res0) # number of residuals
    JJ <- matrix(NA, nrow=mr, ncol=nt) # Initialize the Jacobian
    for (j in 1:nt){
       origPar<-get(theta[j],rho)
       xx <- abs(origPar)
       delta <- if (xx == 0.0) {eps} else { xx*eps }
       ## JN: I prefer eps*(xx + eps)  which is simpler ?? Should we suggest / use a control switch
       prmx<-origPar+delta*dir[j]
       assign(theta[j],prmx,rho)
       res1 <- eval(expr, rho) # new residuals (forward step)
       if (central) { # compute backward step resids for central diff
          prmb <- origPar - dir[j]*delta
          assign(theta[j], prmb, envir=rho) # may be able to make more efficient later??
          resb <- eval(expr, rho)
          JJ[, j] <- dir[j]*(res1-resb)/(2*delta) # vectorized
       } else { ## forward diff
          JJ[,j] <- dir[j]*(res1-res0)/delta
       }  # end forward diff
       assign(theta[j],origPar,rho) # restore the parameter value !! IMPORTANT
    } # end loop over the parameters
    attr(res0, "gradient") <- JJ
    if (ndtrace){
       cat("par:")
       for (j in 1:nt){ cat(get(theta[j],rho)," ") }
       cat("\n")
       print(res0)
    }
    return(res0)
}
```

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

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