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).

```
# 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
```

```
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
```

```
treseval<-microbenchmark(res0<-eval(rexpr, weedenv))
print(treseval)
```

```
## Unit: microseconds
##                          expr   min    lq    mean median     uq    max neval
##  res0 <- eval(rexpr, weedenv) 1.718 1.755 1.97547 1.7895 1.8375 16.851   100
```

```
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) 10.074 10.3705
##      mean  median      uq    max neval
##  10.88016 10.4635 10.7115 39.995   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
##  12.852 13.1435 13.87331 13.4035 13.706 43.788   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) 9.273
##     lq     mean median     uq   max neval
##  9.516 10.15338  9.698 9.9595 37.24   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
##  12.114 12.354 13.08046 12.6855 12.863 40.356   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))
tandnls<-microbenchmark(andnls<-nlsalt::numericDeriv(rexpr, theta, rho=weedenv))
print(tandnls)
```

```
## Unit: microseconds
##                                                          expr    min      lq
##  andnls <- nlsalt::numericDeriv(rexpr, theta, rho = weedenv) 30.727 31.2335
##      mean  median      uq    max neval
##  32.89229 31.6105 32.2055 86.113   100
```

```
## 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
##  40.467 41.0965 42.91185 41.4605 42.1065 94.202   100
```

```
## 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
##  31.288 32.906 34.00766 33.2525 33.6205 76.617   100
```

```
## 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
##  39.853 40.836 43.22993 41.562 42.307 116.972   100
```

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

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

```
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. `model2rjfun()` does much the same work as the `res0<-eval(rexpr, weedenv)` expression evaluation, but adds derivative expressions to the function.

```
# 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
```

```
tres0nlsr<-microbenchmark(res0(start1)) # time it
print(tres0nlsr)
```

```
## Unit: microseconds
##          expr   min     lq    mean median    uq    max neval
##  res0(start1) 6.491 6.7125 7.25533  6.788 6.942 43.569   100
```

```
# or the residuals and jacobian
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) 80.602 82.1755 86.02186 82.905
##      uq     max neval
##  84.626 183.471   100
```

```
## Ways to display information about the residual/jacobian function
# print(funsc); print(funsc(start1)); print(environment(funsc)); print(ls.str(environment(funsc)))
# print(ls(environment(funsc)$data)); eval(eunsc, environment(funsc))
```

```
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) 12.966 13.41 15.43307 13.6815 14.189 63.907   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
jnumd<-jacobian(res0, start1) #  uses default "Richardson" method
jnumd
```

```
##             [,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
tjnumd<-microbenchmark(jnumd<-jacobian(res0, start1))
print(tjnumd)
```

```
## Unit: microseconds
```

```
##                             expr     min       lq     mean median       uq
##  jnumd <- jacobian(res0, start1) 341.022 346.4575 358.0118 350.44 358.9495
##      max neval
##  503.704   100
```

```
jnumds<-jacobian(res0, start1, method="simple") #  uses default "Richardson" method
jnumds
```

```
##            [,1]          [,2]         [,3]
##  [1,] 0.7310586 -1.966066e-01 1.966074e-01
##  [2,] 0.8807971 -1.049923e-01 2.099712e-01
##  [3,] 0.9525741 -4.517645e-02 1.355116e-01
##  [4,] 0.9820138 -1.766267e-02 7.063720e-02
##  [5,] 0.9933071 -6.648052e-03 3.323209e-02
##  [6,] 0.9975274 -2.466509e-03 1.479464e-02
##  [7,] 0.9990889 -9.102211e-04 6.369323e-03
##  [8,] 0.9996646 -3.352377e-04 2.680830e-03
##  [9,] 0.9998766 -1.233794e-04 1.109915e-03
## [10,] 0.9999546 -4.539579e-05 4.537312e-04
## [11,] 0.9999833 -1.670116e-05 1.836115e-04
## [12,] 0.9999939 -6.144063e-06 7.368541e-05
```

```
# Timings of the analytic jacobian calculations
tjnumds<-microbenchmark(jnumds<-jacobian(res0, start1, method="simple"))
print(tjnumds)
```

```
## Unit: microseconds
##                                              expr    min      lq    mean
##  jnumds <- jacobian(res0, start1, method = "simple") 40.246 40.9365 42.2773
##  median      uq    max neval
##  41.173 41.7205 95.219   100
```

```
jnumdc<-jacobian(res0, start1, method="complex") #  uses default "Richardson" method
jnumdc
```

```
##            [,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,] 0.9996646 -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
```

```
# Timings of the analytic jacobian calculations
tjnumdc<-microbenchmark(jnumdc<-jacobian(res0, start1, method="complex"))
print(tjnumdc)
```

```
## Unit: microseconds
##                                              expr    min    lq     mean
##  jnumdc <- jacobian(res0, start1, method = "complex") 48.716 49.79 54.82693
##  median     uq     max neval
```

```
##  50.361 51.454 132.678   100
```

Note that the manual pages for `numDeriv` offer many options for the functions in the package. We have yet to explore many of 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` (.Machine$double.eps^(1/2))‘ndnlsc2‘ : $Central difference with default ‘eps‘(.Machine$double.eps^(1 `ndnlscx`: Forward difference with smaller eps=1e-10 `ndnlscx2`: Central difference with smaller eps=1e-10

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

```
## Matrix comparisons -- uncomment code to show these, which use page space
# attr(ndnls, "gradient")-attr(vfunsc,"gradient")
# attr(ndnlsc, "gradient")-attr(vfunsc,"gradient")
# attr(ndnlsx, "gradient")-attr(vfunsc,"gradient")
# attr(ndnlscx, "gradient")-attr(vfunsc,"gradient")
# jnumd - attr(vfunsc,"gradient")
# jnumds - attr(vfunsc,"gradient")
# jnumdc - attr(vfunsc,"gradient")

## Summary comparisons - maximum absolute differences
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(jnumd - attr(vfunsc,"gradient")))
```

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

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

```
## [1] 1.839972e-05
```

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

```
## [1] 2.775558e-17
```

Observations

Some particular notes:

- the mean time for the default `numericDeriv()` of `nls()` is quite fast and its coefficient of variation (sd/mean) is around 0.42. The timings are actually very slightly faster than the analytic expressions of `nlsr`, but the latter has a COV of 0.36.
- this default method Jacobian unfortunately deviates from the analytical computation by a relatively large amount (of the order of 1e-6 for our example).
- the central difference version of `numericDeriv()` does better (about three orders of magnitude smaller deviation from the analytic result), and the time is comparable with the analytic evaluation.
- making the `eps` parameter smaller degrades the accuracy of the Jacobian computed via `numericDeriv()`. This may be counter-intuitive for those unfamiliar with numerical methods. Essentially, a smaller `eps` results in subtraction of very close values for the residuals.
- the "simple" option of the `numDeriv` function `jacobian()` gives similarly poor accuracy.
- the "Richardson" (default) results of `numDeriv` are of similar accuracy to the central difference option of `numericDeriv()` but at a much greater time cost – about 23 times slower.
- on the other hand, the "complex" option gets an essentially analytic result, approximately 8 orders of magnitude better than the central difference approximation of `numericDeriv`, for a time cost only 3.5 times as great. Unfortunately, not all models are amenable to the complex step approximation.

Cautionary notes on performance results

The results here have been evaluated on a single computer. In fact, while we could process the Rmarkdown file on any of several machines, the work was mainly carried out on a machine characterized with the string

`M21:john-Linux-5.11.0-25-generic|Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz|33482145792bytesRAM`

This is a relatively capable tower PC, but otherwise unremarkable.

Nevertheless, we found that running the timings more than once when other tasks were in progress did result in variations in the mean and standard deviation of the timings of several percentage points. We would expect both absolute differences in times and changes in relative performance with different processors and operating systems, and had thought to carry out some investigation of such differences. However, such effort seems less valuable than pursuing more capable nonlinear least squares and derivative code.

Some notes on 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 labeling 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 approximation to evaluate outside the constraints. Things are decidedly nastier if we do have inadmissible values of the parameters. This is where analytic Jacobian evaluation is very helpful.

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.

```
# 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
```

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

```
## nlsr object: x
## residual sumsquares =  0.050672  on  20 observations
##     after  5    Jacobian and  6 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                1.00955          NA          NA         NA  -1.658e-11       501.9
## b               0.950727          NA          NA         NA   1.262e-11       25.59
## c                1.40371          NA          NA         NA  -1.124e-12   3.823e-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= 869.71  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 = "numericl
##   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= 869.71  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
```

?? this does not give a very clear example.

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)
}
```

References

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.