nlmrt-vignette

John C. Nash

August 13, 2012

Background

This vignette discusses the R package `nlmrt`, that aims to provide computationally robust tools for nonlinear least squares problems. Note that R already has the `nls()` function to solve nonlinear least squares problems, and this function has a large repertoire of tools for such problems. However, it is specifically NOT indicated for problems where the residuals are small or zero. Furthermore, it frequently fails to find a solution if starting parameters are provided that are not close enough to a solution. The tools of `nlmrt` are very much intended to cope with both these issues.

The functions are also intended to provide stronger support for bounds constraints and to introduce the capability for **masks**, that is, parameters that are fixed for a given run of the function.

`nlmrt` tools generally do not return the large nls-style object. However, we do provide a tool `wrapnls` that will run either `nlxb` followed by a call to `nls`. The call to `nls` is adjusted to use the `port` algorithm if there are bounds constraints.

1 An example problem and its solution

Let us try an example initially presented by (Ratkowsky 1983) and developed by (Huet et˜al. 1996). This is a model for the regrowth of pasture. We set up the computation by putting the data for the problem in a data frame, and specifying the formula for the model. This can be as a formula object, but I have found that saving it as a character string seems to give fewer difficulties. Note the "˜" that implies "is modeled by". There must be such an element in the formula for this package (and for `nls()`). We also specify two sets of starting parameters, that is, the `ones` which is a trivial (but possibly unsuitable) start with all parameters set to 1, and `huetstart` which was suggested in (Huet et˜al. 1996). Finally we load the routines in the package `nlmrt`.

```
> options(width=60)
> pastured <- data.frame(
+ time=c(9, 14, 21, 28, 42, 57, 63, 70, 79),
```

```
+ yield= c(8.93, 10.8, 18.59, 22.33, 39.35,
+          56.11, 61.73, 64.62, 67.08))
> regmod <- "yield ~ t1 - t2*exp(-exp(t3+t4*log(time)))"
> ones <- c(t1=1, t2=1, t3=1, t4=1) # all ones start
> huetstart <- c(t1=70, t2=60, t3=0, t4=1)
> require(nlmrt)
```

Let us now call the routine `nlsmnqb` (even though we are not specifying bounds). We try both starts.

```
> anmrt <- nlxb(regmod, start=ones, trace=FALSE, data=pastured)
> print(anmrt)
$resid
[1]  0.48069948  0.66930970 -2.28432650  0.84373846
[5]  0.73457526  0.06655466 -0.98580893 -0.02505846
[9]  0.50031634

$jacobian
      t1          t2        t3        t4
 [1,]  1 -0.98156716  1.126420  2.474999
 [2,]  1 -0.94819229  3.111329  8.210975
 [3,]  1 -0.86978356  7.484690 22.787306
 [4,]  1 -0.75843621 12.934908 43.101760
 [5,]  1 -0.48427212 21.659422 80.955765
 [6,]  1 -0.22338362 20.652294 83.498282
 [7,]  1 -0.14933159 17.515486 72.569018
 [8,]  1 -0.08690194 13.094925 55.633728
 [9,]  1 -0.03850206  7.735031 33.797814

$feval
[1] 76

$jeval
[1] 50

$coeffs
[1] 69.955179 61.681444 -9.208935  2.377819

$ssquares
[1] 8.375884


> anmrtx <- try(nlxb(regmod, start=huetstart, trace=FALSE, data=pastured))
> print(strwrap(anmrtx))
 [1] "c(0.480699476110992, 0.669309701586503,"
 [2] "-2.28432650017661, 0.843738460841614,"
```

```
 [3] "0.734575256138093, 0.0665546618861583,"
 [4] "-0.985808933151056, -0.0250584603521418,"
 [5] "0.500316337120296)"
 [6] "c(1, 1, 1, 1, 1, 1, 1, 1, 1, -0.981567160420883,"
 [7] "-0.948192289406167, -0.869783557170751,"
 [8] "-0.758436212560273, -0.484272123696113,"
 [9] "-0.223383622127412, -0.149331587423979,"
[10] "-0.0869019449646661, -0.0385020596618461,"
[11] "1.12642043233262, 3.11132895498809, 7.48468988716119,"
[12] "12.9349083313689, 21.6594224095687, 20.652293670436,"
[13] "17.51548586967, 13.0949252904654, 7.73503096811733,"
[14] "2.47499865833493, 8.2109754835055, 22.7873063008638,"
[15] "43.1017598804902, 80.9557650898109, 83.4982821079476,"
[16] "72.56901775625, 55.6337277915341, 33.7978144524062)"
[17] "61"
[18] "39"
[19] "c(69.9551789601637, 61.6814436396711,"
[20] "-9.20893535565824, 2.37781880027694)"
[21] "8.37588355893792"
```

Note that the standard `nls()` of R fails to find a solution from either start.

```
> anls <- try(nls(regmod, start=ones, trace=FALSE, data=pastured))
> print(strwrap(anls))
[1] "Error in nlsModel(formula, mf, start, wts) : singular"
[2] "gradient matrix at initial parameter estimates"

> anlsx <- try(nls(regmod, start=huetstart, trace=FALSE, data=pastured))
> print(strwrap(anlsx))
[1] "Error in nls(regmod, start = huetstart, trace ="
[2] "FALSE, data = pastured) : singular gradient"
```

In both cases, the `nls()` failed with a 'singular gradient'. This implies the Jacobian is effectively singular at some point. The Levenberg-Marquardt stabilization used in `nlxb` avoids this particular issue by augmenting the Jacobian until it is non-singular. The details of this common approach may be found elsewhere (Nash 1979). ?? Do we want a page ref?

There are some other tools for R that aim to solve nonlinear least squares problems. We have not yet been able to successfully use the INRA package `nls2`. This is a quite complicated package and is not installable as a regular R package using `install.packages()`. Note that there is a very different package by the same name on CRAN by Gabor Grothendieck.

2 The `nls` solution

We can call `nls` after getting a potential nonlinear least squares solution using `nlxb`. Package `nlmrt` has function `wrapnls` to allow this to be carried out automatically. Thus,

```
  > awnls <- wrapnls(regmod, start=ones, data=pastured)
  > print(awnls)
Nonlinear regression model
  model:  yield ~ t1 - t2 * exp(-exp(t3 + t4 * log(time)))
   data:  data
    t1     t2     t3     t4
69.955 61.681 -9.209  2.378
 residual sum-of-squares: 8.376

Number of iterations to convergence: 0
Achieved convergence tolerance: 8.334e-08
  > cat("Note that the above is just the nls() summary result.\n")
Note that the above is just the nls() summary result.
```

3 Problems specified by residual functions

The model expressions in R , such as

`yield ~ t1 - t2*exp(-exp(t3+t4*log(time)))`

are an extremely helpful feature of the language. Moreover, they are used to compute symbolic or automatic derivatives, so we do not have to rely on numerical approximations for the Jacobian of the nonlinar least squares problem. However, there are many situations where the expression structure is not flexible enough to allow us to define our residuals, or where the construction of the residuals is simply too complicated. In such cases it is helpful to have tools that work with R functions.

Once we have an R function for the residuals, we can use the safeguarded Marquardt routine `nlfb` from package `nlmrt` or else the routine `nls.lm` from package `minpack.lm` (Elzhov, Mullen, Spiess, and Bolker 2012). The latter is built on the Minpack Fortran codes of (Moré, Garbow, and Hillstrom 1980) implemented by Kate Mullen. `nlfb` is written entirely in R , and is intended to be quite aggessive in ensuring it finds a good minimum. Thus these two approaches have somewhat different characteristics.

Let us consider a slightly different problem, called WEEDS. Here the objective is to model a set of 12 data points (density y of weeds at annual time points tt) versus the time index. (A minor note: use of `t` rather than `tt` in R may encourage confusion with the transpose function `t()`, so I tend to avoid plain `t`.) The model suggested was a 3-parameter logistic function,

$y_{model} = b_1/(1 + b_2 exp(-b_3 tt))$

and while it is possible to use this formulation, a scaled version gives slightly better results

$y_{model} = 100b_1/(1 + 10b_2 exp(-0.1b_3 tt))$

The residuals for this latter model (in form "model" minus "data") are coded in R in the following code chunk in the function `shobbs.res`. We have also coded the Jacobian for this model as `shobbs.jac`

```
  > shobbs.res <- function(x){ # scaled Hobbs weeds problem -- residual
  + # This variant uses looping
```

```
  +     if(length(x) != 3) stop("hobbs.res -- parameter vector n!=3")
  +     y <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948,
  +          75.995, 91.972)
  +     tt <- 1:12
  +     res <- 100.0*x[1]/(1+x[2]*10.*exp(-0.1*x[3]*tt)) - y
  + }
  > shobbs.jac <- function(x) { # scaled Hobbs weeds problem -- Jacobian
  +     jj <- matrix(0.0, 12, 3)
  +     tt <- 1:12
  +     yy <- exp(-0.1*x[3]*tt) # We don't need data for the Jacobian
  +     zz <- 100.0/(1+10.*x[2]*yy)
  +     jj[tt,1]  <-  zz
  +     jj[tt,2]  <-  -0.1*x[1]*zz*zz*yy
  +     jj[tt,3]  <-  0.01*x[1]*zz*zz*yy*x[2]*tt
  +     return(jj)
  + }
```

With package `nlmrt`, function `nlfb` can be used to estimate the parameters of the WEEDS problem as follows, where we use the naive starting point where all parameters are 1.

```
  > st <- c(b1=1, b2=1, b3=1)
  > ans1 <- nlfb(st, shobbs.res, shobbs.jac, trace=FALSE)
  > print(ans1)
$resid
 [1]  0.01189993 -0.03275547  0.09202996  0.20878182
 [5]  0.39263404 -0.05759436 -1.10572842  0.71578576
 [9] -0.10764762 -0.34839635  0.65259251 -0.28756791

$jacobian
           [,1]      [,2]       [,3]
 [1,]  2.711658 -1.054282  0.5175642
 [2,]  3.673674 -1.414187  1.3884948
 [3,]  4.959588 -1.883714  2.7742382
 [4,]  6.664474 -2.485844  4.8813666
 [5,]  8.900539 -3.240359  7.9537273
 [6,] 11.792062 -4.156794 12.2438293
 [7,] 15.463505 -5.224121 17.9522463
 [8,] 20.018622 -6.398588 25.1293721
 [9,] 25.510631 -7.594104 33.5526319
[10,] 31.908250 -8.682775 42.6251654
[11,] 39.068787 -9.513292 51.3725387
[12,] 46.733360 -9.948174 58.6046596

$feval
[1] 24

$jeval
[1] 15

$coeffs
[1] 1.961863 4.909164 3.135697

$ssquares
[1] 2.587277
```

This works very well, with almost identical iterates as given by `nlxb`. (Since the algorithms are the same, this should be the case.) Note that we turn off

the `trace` output. There is also the possibility of interrupting the iterations to `watch` the progress. Changing the value of `watch` in the call to `nlfb` below allows this. In this code chunk, we use an internal numerical approximation to the Jacobian.

```
  > cat("No jacobian function -- use internal approximation\n")
No jacobian function -- use internal approximation
  > ans1n <- nlfb(st, shobbs.res, trace=FALSE, control=list(watch=FALSE)) # NO jacfn
  > print(ans1n)
$resid
 [1]  0.01189993 -0.03275547  0.09202996  0.20878182
 [5]  0.39263405 -0.05759436 -1.10572842  0.71578576
 [9] -0.10764762 -0.34839635  0.65259251 -0.28756790

$jacobian
           [,1]      [,2]       [,3]
 [1,]  2.711658 -1.054282  0.5175643
 [2,]  3.673674 -1.414186  1.3884948
 [3,]  4.959588 -1.883714  2.7742383
 [4,]  6.664474 -2.485844  4.8813668
 [5,]  8.900539 -3.240359  7.9537278
 [6,] 11.792062 -4.156793 12.2438302
 [7,] 15.463505 -5.224120 17.9522477
 [8,] 20.018622 -6.398587 25.1293740
 [9,] 25.510631 -7.594104 33.5526342
[10,] 31.908250 -8.682774 42.6251678
[11,] 39.068787 -9.513291 51.3725408
[12,] 46.733360 -9.948173 58.6046604

$feval
[1] 29

$jeval
[1] 15

$coeffs
[1] 1.961863 4.909164 3.135697

$ssquares
[1] 2.587277
```

Note that we could also form the sum of squares function and the gradient and use a function minimization code. The next code block shows how this is done, creating the sum of squares function and its gradient, then using the `optimx` package to call a number of minimizers simultaneously.

```
  > shobbs.f <- function(x){
  +    res <- shobbs.res(x)
  +    as.numeric(crossprod(res))
  + }
  > shobbs.g <- function(x){
  +    res <- shobbs.res(x) # This is NOT efficient -- we generally have res already calcula
  +    JJ <- shobbs.jac(x)
  +    2.0*as.vector(crossprod(JJ,res))
  + }
  > require(optimx)
  > aopx <- optimx(st, shobbs.f, shobbs.g, control=list(all.methods=TRUE))
```

```
end topstuff in optimxCRAN
```

```
> optansout(aopx, NULL) # no file output
```

```
                        par
2  1.911961, 4.824629, 3.158554
3  1.964498, 4.911596, 3.133976
7  1.961833, 4.909121, 3.135712
5  1.961867, 4.909168, 3.135695
1  1.961863, 4.909165, 3.135698
12 1.961863, 4.909165, 3.135697
11 1.961863, 4.909164, 3.135697
4  1.961863, 4.909164, 3.135697
10 1.961863, 4.909164, 3.135697
6  1.961863, 4.909164, 3.135697
9  1.961863, 4.909164, 3.135697
8  1.961863, 4.909164, 3.135697
    fvalues      method  fns grs itns conv  KKT1 KKT2
2  2.667931          CG  427 101 NULL    1 FALSE TRUE
3  2.587651 Nelder-Mead  196  NA NULL    0 FALSE TRUE
7  2.587277         spg  188  NA  150    0  TRUE TRUE
5  2.587277         nlm   NA  NA   50    0  TRUE TRUE
1  2.587277        BFGS  119  36 NULL    0  TRUE TRUE
12 2.587277      bobyqa  705  NA NULL    0  TRUE TRUE
11 2.587277      newuoa 1957  NA NULL    0  TRUE TRUE
4  2.587277    L-BFGS-B   41  41 NULL    0  TRUE TRUE
10 2.587277      Rvmmin   83  47 NULL    0  TRUE TRUE
6  2.587277      nlminb   31  29   28    0  TRUE TRUE
9  2.587277      Rcgmin  138  50 NULL    0  TRUE TRUE
8  2.587277      ucminf   46  46 NULL    0  TRUE TRUE
   xtimes
2   0.012
3   0.004
7   0.044
5   0.004
1   0.008
12   0.02
11  0.056
4   0.004
10  0.012
6   0.004
9   0.008
8   0.004
[1] TRUE
```

```
> cat("\nNow with numerical gradient approximation or derivative free methods\n")
```

```
Now with numerical gradient approximation or derivative free methods
```

```
> aopxn <- optimx(st, shobbs.f, control=list(all.methods=TRUE))
```

```
end topstuff in optimxCRAN
function(x){
   res <- shobbs.res(x)
   as.numeric(crossprod(res))
}
```

```
> optansout(aopxn, NULL) # no file output
```

```
                        par
2  1.799764, 4.597043, 3.207810
3  1.964498, 4.911596, 3.133976
8  1.961940, 4.909044, 3.135611
7  1.961852, 4.909111, 3.135695
1  1.961897, 4.909192, 3.135676
10 1.961870, 4.909152, 3.135689
4  1.961876, 4.909172, 3.135688
5  1.961867, 4.909168, 3.135695
12 1.961863, 4.909165, 3.135697
11 1.961863, 4.909164, 3.135697
9  1.961863, 4.909164, 3.135697
6  1.961863, 4.909164, 3.135697
    fvalues      method  fns grs itns conv  KKT1 KKT2
```

```
2  3.829886          CG  413 101 NULL    1 FALSE TRUE
3  2.587651 Nelder-Mead  196  NA NULL    0 FALSE TRUE
8  2.587278      ucminf   45  45 NULL    0 FALSE TRUE
7  2.587277         spg  174  NA  135    0  TRUE TRUE
1  2.587277        BFGS  118  36 NULL    0  TRUE TRUE
10 2.587277      Rvmmin   83  44 NULL    0  TRUE TRUE
4  2.587277    L-BFGS-B   45  45 NULL    0  TRUE TRUE
5  2.587277         nlm   NA  NA   50    0  TRUE TRUE
12 2.587277      bobyqa  705  NA NULL    0  TRUE TRUE
11 2.587277      newuoa 1957  NA NULL    0  TRUE TRUE
9  2.587277      Rcgmin  128  48 NULL    0  TRUE TRUE
6  2.587277      nlminb   32  93   27    0  TRUE TRUE
   xtimes
2    0.02
3   0.004
8   0.004
7   0.032
1   0.008
10  0.016
4   0.008
5   0.004
12  0.016
11  0.056
9   0.076
6   0.004
[1] TRUE
```

We see that most of the minimizers work with either the analytic or approximated gradient. The 'CG' option of function `optim()` does not do very well in either case. As the author of the original step and description and then Turbo Pascal code, I can say I was never very happy with this method and replaced it recently with `Rcgmin` from the package of the same name, in the process adding the possibility of bounds or masks constraints.

4 Converting an expression to a function

Clearly if we have an expression, it would be nice to be able to automatically convert this to a function, if possible also getting the derivatives. Indeed, it is possible to convert an expression to a function, and there are several ways to do this (references??). In package `nlmrt` we provide the tools `model2grfun.R`, `model2jacfun.R`, `model2resfun.R`, and `model2ssfun.R` to convert a model expression to a function to compute the gradient, Jacobian, residuals or sum of squares functions respectively. We do not provide any tool for converting a function for the residuals back to an expression, as functions can use structures that are not easily expressed as R expressions.

Below are code chunks to illustrate the generation of the residual, sum of squares, Jacobian and gradient code for the Ratkowsky problem used earlier in the vignette. The commented-out first line shows how we would use one of these function generators to output the function to a file named "testresfn.R". However, it is not necessary to generate the file.

First, let us generate the residuals. We must supply the names of the parameters, and do this via the starting vector of parameters `ones`. The actual values are not needed by `model2resfun`, just the names. Other names are drawn from the variables used in the model expression `regmod`.

```
  > # jres <- model2resfun(regmod, ones, funname="myxres", file="testresfn.R")
  > jres <- model2resfun(regmod, ones)
  > print(jres)
function (prm, yield = NULL, time = NULL)
{
    t1 <- prm[[1]]
    t2 <- prm[[2]]
    t3 <- prm[[3]]
    t4 <- prm[[4]]
    resids <- as.numeric(eval(t1 - t2 * exp(-exp(t3 + t4 * log(time))) -
        yield))
}
<environment: 0x94024b4>
  > valjres <- jres(ones, yield=pastured$yield, time=pastured$time)
  > cat("valjres:")
valjres:
  > print(valjres)
[1]  -7.93  -9.80 -17.59 -21.33 -38.35 -55.11 -60.73 -63.62
[9] -66.08
```

Now let us also generate the Jacobian and test it using the numerical approximations from package `numDeriv`.

```
  > jjac <- model2jacfun(regmod, ones)
  > print(jjac)
function (prm, yield = NULL, time = NULL)
{
    t1 <- prm[[1]]
    t2 <- prm[[2]]
    t3 <- prm[[3]]
    t4 <- prm[[4]]
    localdf <- data.frame(yield, time)
    jstruc <- with(localdf, eval({
        .expr1 <- log(time)
        .expr4 <- exp(t3 + t4 * .expr1)
        .expr6 <- exp(-.expr4)
        .value <- t1 - t2 * .expr6 - yield
        .grad <- array(0, c(length(.value), 4), list(NULL, c("t1",
            "t2", "t3", "t4")))
        .grad[, "t1"] <- 1
        .grad[, "t2"] <- -.expr6
        .grad[, "t3"] <- t2 * (.expr6 * .expr4)
        .grad[, "t4"] <- t2 * (.expr6 * (.expr4 * .expr1))
        attr(.value, "gradient") <- .grad
        .value
    }))
    jacmat <- attr(jstruc, "gradient")
    return(jacmat)
}
<environment: 0x9136478>
  > # Note that we now need some data!
  > valjjac <- jjac(ones, yield=pastured$yield, time=pastured$time)
  > cat("valjac:")
valjac:
  > print(valjjac)
     t1           t2          t3          t4
 [1,]  1 -2.372394e-11 5.803952e-10 1.275259e-09
 [2,]  1 -2.968334e-17 1.129628e-15 2.981152e-15
 [3,]  1 -1.617220e-25 9.231728e-24 2.810620e-23
```

```
 [4,]  1 -8.811009e-34 6.706226e-32 2.234652e-31
 [5,]  1 -2.615402e-50 2.985948e-48 1.116049e-47
 [6,]  1 -5.122907e-68 7.937538e-66 3.209187e-65
 [7,]  1 -4.229682e-75 7.243404e-73 3.001040e-72
 [8,]  1 -2.304433e-83 4.384869e-81 1.862910e-80
 [9,]  1 -5.467023e-94 1.174012e-91 5.129784e-91
```

```
> # Now compute the numerical approximation
> Jn <- jacobian(jres, ones, , yield=pastured$yield, time=pastured$time)
> cat("maxabsdiff=",max(abs(Jn-valjjac)),"\n")
```

```
maxabsdiff= 3.774395e-10
```

As with the WEEDS problem, we can compute the sum of squares function and the gradient.

```
> ssfn <- model2ssfun(regmod, ones) # problem getting the data attached!
> print(ssfn)
```

```
function (prm, yield = NULL, time = NULL)
{
    t1 <- prm[[1]]
    t2 <- prm[[2]]
    t3 <- prm[[3]]
    t4 <- prm[[4]]
    resids <- as.numeric(eval(t1 - t2 * exp(-exp(t3 + t4 * log(time))) -
        yield))
    ss <- as.numeric(crossprod(resids))
}
<environment: 0x9d0f324>
```

```
> valss <- ssfn(ones, yield=pastured$yield, time=pastured$time)
> cat("valss: ",valss,"\n")
```

```
valss:  17533.34
```

```
> grfn <- model2grfun(regmod, ones) # problem getting the data attached!
> print(grfn)
```

```
function (prm, yield = NULL, time = NULL)
{
    t1 <- prm[[1]]
    t2 <- prm[[2]]
    t3 <- prm[[3]]
    t4 <- prm[[4]]
    localdf <- data.frame(yield, time)
    jstruc <- with(localdf, eval({
        .expr1 <- log(time)
        .expr4 <- exp(t3 + t4 * .expr1)
        .expr6 <- exp(-.expr4)
        .value <- t1 - t2 * .expr6 - yield
        .grad <- array(0, c(length(.value), 4), list(NULL, c("t1",
            "t2", "t3", "t4")))
        .grad[, "t1"] <- 1
        .grad[, "t2"] <- -.expr6
        .grad[, "t3"] <- t2 * (.expr6 * .expr4)
        .grad[, "t4"] <- t2 * (.expr6 * (.expr4 * .expr1))
        attr(.value, "gradient") <- .grad
        .value
    }))
    jacmat <- attr(jstruc, "gradient")
    resids <- as.numeric(eval(t1 - t2 * exp(-exp(t3 + t4 * log(time))) -
        yield))
    grj <- as.vector(2 * crossprod(jacmat, resids))
}
<environment: 0x9d58734>
```

```
> valgr <- grfn(ones, yield=pastured$yield, time=pastured$time)
> cat("valgr:")
```

```
valgr:
```

```
  > print(valgr)
[1] -6.810800e+02  3.762623e-10 -9.205090e-09 -2.022566e-08
```

```
  > gn <- grad(ssfn, ones, yield=pastured$yield, time=pastured$time)
  > cat("maxabsdiff=",max(abs(gn-valgr)),"\n")
maxabsdiff= 7.476956e-08
```

Moreover, we can use the Huet starting parameters as a double check on our conversion of the expression to various optimization-style functions.

```
  > cat("\n\nHuetstart:")
Huetstart:
```

```
  > print(huetstart)
t1 t2 t3 t4
70 60  0  1
```

```
  > valjres <- jres(huetstart, yield=pastured$yield, time=pastured$time)
  > cat("valjres:")
valjres:
```

```
  > print(valjres)
[1] 61.06260 59.19995 51.41000 47.67000 30.65000 13.89000
[7]  8.27000  5.38000  2.92000
```

```
  > valss <- ssfn(huetstart, yield=pastured$yield, time=pastured$time)
  > cat("valss:", valss, "\n")
valss: 13386.91
```

```
  > valjjac <- jjac(huetstart, yield=pastured$yield, time=pastured$time)
  > cat("valjac:")
valjac:
```

```
  > print(valjjac)
     t1           t2          t3          t4
 [1,]  1 -1.234098e-04 6.664129e-02 1.464259e-01
 [2,]  1 -8.315287e-07 6.984841e-04 1.843340e-03
 [3,]  1 -7.582560e-10 9.554026e-07 2.908745e-06
 [4,]  1 -6.914400e-13 1.161619e-09 3.870753e-09
 [5,]  1 -5.749522e-19 1.448880e-15 5.415433e-15
 [6,]  1 -1.758792e-25 6.015069e-22 2.431923e-21
 [7,]  1 -4.359610e-28 1.647933e-24 6.827607e-24
 [8,]  1 -3.975450e-31 1.669689e-27 7.093665e-27
 [9,]  1 -4.906095e-35 2.325489e-31 1.016110e-30
```

```
  > Jn <- jacobian(jres, huetstart, , yield=pastured$yield, time=pastured$time)
  > cat("maxabsdiff=",max(abs(Jn-valjjac)),"\n")
maxabsdiff= 5.394534e-10
```

```
  > valgr <- grfn(huetstart, yield=pastured$yield, time=pastured$time)
  > cat("valgr:")
valgr:
```

```
  > print(valgr)
[1] 560.90509095  -0.01516998   8.22137957  18.10084037
```

```
  > gn <- grad(ssfn, huetstart, yield=pastured$yield, time=pastured$time)
  > cat("maxabsdiff=",max(abs(gn-valgr)),"\n")
maxabsdiff= 5.952869e-08
```

Now that we have these functions, let us apply them with `nlfb`.

```
  > cat("All ones to start\n")
All ones to start
  > anlfb <- nlfb(ones, jres, jjac, trace=FALSE, yield=pastured$yield, time=pastured$time)
  > print(strwrap(anlfb))
 [1] "c(0.480699475409779, 0.669309701325741,"
 [2] "-2.28432649983562, 0.843738461541676,"
 [3] "0.734575256578069, 0.0665546616416748,"
 [4] "-0.985808933450038, -0.0250584605193325,"
 [5] "0.500316337308163)"
 [6] "c(1, 1, 1, 1, 1, 1, 1, 1, 1, -0.981567160415026,"
 [7] "-0.948192289394349, -0.869783557151951,"
 [8] "-0.758436212539591, -0.484272123689345,"
 [9] "-0.22338362214097, -0.14933158744104,"
[10] "-0.086901944981799, -0.0385020596749348,"
[11] "1.12642043272705, 3.1113289557883, 7.48468988842378,"
[12] "12.9349083327494, 21.6594224104496, 20.6522936715837,"
[13] "17.5154858712384, 13.0949252924535, 7.73503097021314,"
[14] "2.47499865920158, 8.21097548561731, 22.7873063047078,"
[15] "43.1017598850905, 80.9557650931036, 83.498282112588,"
[16] "72.569017762748, 55.6337277999807, 33.7978144615637)"
[17] "74"
[18] "48"
[19] "c(69.9551789612429, 61.6814436418531,"
[20] "-9.20893535490747, 2.37781880008123)"
[21] "8.37588355893788"
  > cat("Huet start\n")
Huet start
  > anlfbh <- nlfb(huetstart, jres, jjac, trace=FALSE, yield=pastured$yield, time=pastured$t
  > print(strwrap(anlfbh))
 [1] "c(0.480699465869456, 0.669309697775223,"
 [2] "-2.28432649519877, 0.84373847107085,"
 [3] "0.734575262591456, 0.0665546583437617,"
 [4] "-0.985808937499776, -0.0250584627932966,"
 [5] "0.500316339841277)"
 [6] "c(1, 1, 1, 1, 1, 1, 1, 1, 1, -0.981567160335378,"
 [7] "-0.94819228923362, -0.869783556896137,"
 [8] "-0.75843621225793, -0.484272123596337,"
 [9] "-0.223383622324199, -0.149331587672017,"
[10] "-0.0869019452139657, -0.0385020598524092,"
[11] "1.12642043808933, 3.11132896666899, 7.48468990559557,"
[12] "12.9349083515304, 21.6594224224275, 20.652293687139,"
[13] "17.5154858924942, 13.0949253194057, 7.73503099863509,"
[14] "2.47499867098372, 8.21097551433206, 22.7873063569877,"
[15] "43.1017599476725, 80.9557651378729, 83.498282175479,"
[16] "72.5690178508139, 55.6337279144867, 33.7978145857519)"
[17] "60"
[18] "37"
[19] "c(69.9551789758633, 61.6814436714725,"
[20] "-9.20893534470294, 2.37781879742191)"
[21] "8.37588355893793"
```

5 Using bounds and masks

The manual for `nls()` tells us that bounds are restricted to the 'port' algorithm.

```
lower, upper: vectors of lower and upper bounds, replicated to be as
          long as 'start'.  If unspecified, all parameters are assumed
          to be unconstrained.  Bounds can only be used with the
```

```
'"port"' algorithm.  They are ignored, with a warning, if
given for other algorithms.
```

Later in the manual, there is the discomforting warning:

```
The 'algorithm = "port"' code appears unfinished, and does not
even check that the starting value is within the bounds.  Use with
caution, especially where bounds are supplied.
```

We will base the rest of this discussion on the examples in man/nlmrt-package.Rd, and use an unscaled version of the WEEDS problem.

First, let us estimate the model with no constraints.

```
> require(nlmrt)
> # 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)
> tdat <- 1:length(ydat)
> weeddata1 <- data.frame(y=ydat, tt=tdat)
> start1 <- c(b1=1, b2=1, b3=1) # name parameters for nlxb, nls, wrapnls.
> eunsc <-  y ~ b1/(1+b2*exp(-b3*tt))
> anlxb1 <- try(nlxb(eunsc, start=start1, data=weeddata1))
> print(anlxb1)
$resid
 [1]  0.01189993 -0.03275547  0.09202996  0.20878182
 [5]  0.39263404 -0.05759436 -1.10572842  0.71578576
 [9] -0.10764762 -0.34839635  0.65259251 -0.28756791

$jacobian
             b1         b2         b3
 [1,] 0.02711658 -0.1054282   5.175642
 [2,] 0.03673674 -0.1414187  13.884948
 [3,] 0.04959588 -0.1883714  27.742382
 [4,] 0.06664474 -0.2485844  48.813666
 [5,] 0.08900539 -0.3240359  79.537273
 [6,] 0.11792062 -0.4156794 122.438293
 [7,] 0.15463505 -0.5224121 179.522463
 [8,] 0.20018622 -0.6398588 251.293721
 [9,] 0.25510631 -0.7594104 335.526319
[10,] 0.31908250 -0.8682775 426.251654
[11,] 0.39068787 -0.9513292 513.725387
[12,] 0.46733360 -0.9948174 586.046596

$feval
[1] 36

$jeval
[1] 22

$coeffs
[1] 196.1862618  49.0916395   0.3135697

$ssquares
[1] 2.587277
```

Now let us see if we can apply bounds. Note that we name the parameters in the vectors for the bounds. First we apply bounds that are NOT active at the unconstrained solution.

```
  > # WITH BOUNDS
  > startf1 <- c(b1=1, b2=1, b3=.1) # a feasible start when b3 <= 0.25
  > anlxb1 <- try(nlxb(eunsc, start=startf1, lower=c(b1=0, b2=0, b3=0),
  +       upper=c(b1=500, b2=100, b3=5), data=weeddata1))
  > print(anlxb1)
$resid
 [1]  0.01189993 -0.03275547  0.09202996  0.20878182
 [5]  0.39263404 -0.05759436 -1.10572842  0.71578576
 [9] -0.10764762 -0.34839635  0.65259251 -0.28756791

$jacobian
             b1         b2         b3
 [1,] 0.02711658 -0.1054282   5.175642
 [2,] 0.03673674 -0.1414187  13.884948
 [3,] 0.04959588 -0.1883714  27.742382
 [4,] 0.06664474 -0.2485844  48.813666
 [5,] 0.08900539 -0.3240359  79.537273
 [6,] 0.11792062 -0.4156794 122.438293
 [7,] 0.15463505 -0.5224121 179.522463
 [8,] 0.20018622 -0.6398588 251.293721
 [9,] 0.25510631 -0.7594104 335.526319
[10,] 0.31908250 -0.8682775 426.251654
[11,] 0.39068787 -0.9513292 513.725387
[12,] 0.46733360 -0.9948174 586.046596

$feval
[1] 29

$jeval
[1] 17

$coeffs
[1] 196.1862618  49.0916395   0.3135697

$ssquares
[1] 2.587277
```

We note that `nls()` also solves this case.

```
  > anlsb1 <- try(nls(eunsc, start=startf1, lower=c(b1=0, b2=0, b3=0),
  +      upper=c(b1=500, b2=100, b3=5), data=weeddata1, algorithm='port'))
  > print(anlsb1)
Nonlinear regression model
  model:  y ~ b1/(1 + b2 * exp(-b3 * tt))
   data:  weeddata1
      b1       b2       b3
196.1863  49.0916   0.3136
 residual sum-of-squares: 2.587

Algorithm "port", convergence message: relative convergence (4)
```

Now we will change the bounds so the start is infeasible.

```
  > ## Uncon solution has bounds ACTIVE. Infeasible start
  > anlxb2i <- try(nlxb(eunsc, start=start1, lower=c(b1=0, b2=0, b3=0),
  +             upper=c(b1=500, b2=100, b3=.25), data=weeddata1))
  > print(anlxb2i)
[1] "Error in nlxb(eunsc, start = start1, lower = c(b1 = 0, b2 = 0, b3 = 0),  : \n  Infeasible start\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in nlxb(eunsc, start = start1, lower = c(b1 = 0, b2 = 0, b3 = 0),     upper = c(b1 = 500, b2 = 100, b3 = 0.25), data
```

```
  > anlsb2i <- try(nls(eunsc, start=start1, lower=c(b1=0, b2=0, b3=0),
  +             upper=c(b1=500, b2=100, b3=.25), data=weeddata1, algorithm='port'))
  > print(anlsb2i)
[1] "Error in nls(eunsc, start = start1, lower = c(b1 = 0, b2 = 0, b3 = 0),  : \n  Convergence failure: initial par violates const
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in nls(eunsc, start = start1, lower = c(b1 = 0, b2 = 0, b3 = 0),     upper = c(b1 = 500, b2 = 100, b3 = 0.25), data =
```

Both `nlxb()` and `nls()` (with 'port') do the right thing and refuse to proceed. There is a minor "glitch" in the output processing of both `knitR` and `Sweave` here. Let us start them off properly and see what they accomplish.

```
  > ## Uncon solution has bounds ACTIVE. Feasible start
  > anlxb2f <- try(nlxb(eunsc, start=startf1, lower=c(b1=0, b2=0, b3=0),
  +    upper=c(b1=500, b2=100, b3=.25), data=weeddata1))
  > print(anlxb2f)
$resid
 [1]  1.8873141  1.9613574  2.1153439  2.1255016  2.0179261
 [6]  1.0532232 -0.7344513  0.1965226 -1.4660939 -2.1115581
[11] -0.4888194  0.9924830

$jacobian
      b1         b2 b3
 [1,]  0 -0.0806410  0
 [2,]  0 -0.1027038  0
 [3,]  0 -0.1305065  0
 [4,]  0 -0.1653582  0
 [5,]  0 -0.2087536  0
 [6,]  0 -0.2623256  0
 [7,]  0 -0.3277429  0
 [8,]  0 -0.4065236  0
 [9,]  0 -0.4997414  0
[10,]  0 -0.6076051  0
[11,]  0 -0.7289289  0
[12,]  0 -0.8605590  0

$feval
[1] 32

$jeval
[1] 16

$coeffs
[1] 500.00000  87.94248   0.25000

$ssquares
[1] 29.99273
  > anlsb2f <- try(nls(eunsc, start=startf1, lower=c(b1=0, b2=0, b3=0),
  +    upper=c(b1=500, b2=100, b3=.25), data=weeddata1, algorithm='port'))
  > print(anlsb2f)
Nonlinear regression model
  model:  y ~ b1/(1 + b2 * exp(-b3 * tt))
   data:  weeddata1
    b1     b2     b3
500.00  87.94   0.25
 residual sum-of-squares: 29.99

Algorithm "port", convergence message: both X-convergence and relative convergence (5)
```

Both methods get essentially the same answer for the bounded problem, and this solution has parameters `b1` and `b3` at their upper bounds. The Jacobian elements for these parameters are zero as returned by `nlxb()`.

Let us now turn to **masks**, which functions from `nlmrt` are designed to handle. Masks are also available with packages `Rcgmin` and `Rvmmin`. I would like to hear if other packages offer this capability.

```
  > ## TEST MASKS
  > anlsmnqm <- try(nlxb(eunsc, start=start1, lower=c(b1=0, b2=0, b3=0),
  +    upper=c(b1=500, b2=100, b3=5), masked=c("b2"), data=weeddata1))
  > print(anlsmnqm) # b2 masked
$resid
 [1]  22.387335  22.901373  22.856287  21.850150  19.708920
 [6]  15.468341   8.910728   3.298869  -6.980585 -18.627939
[11] -30.690198 -45.826592

$jacobian
            b1 b2       b3
 [1,] 0.5494911  0 12.47699
 [2,] 0.5980219  0 24.23234
 [3,] 0.6447051  0 34.63517
 [4,] 0.6887881  0 43.21632
 [5,] 0.7296948  0 49.70632
 [6,] 0.7670431  0 54.03717
 [7,] 0.8006408  0 56.31420
 [8,] 0.8304640  0 56.76997
 [9,] 0.8566247  0 55.71260
[10,] 0.8793351  0 53.47875
[11,] 0.8988729  0 50.39696
[12,] 0.9155510  0 46.76318

$feval
[1] 57

$jeval
[1] 33

$coeffs
[1] 50.4017868  1.0000000  0.1986149

$ssquares
[1] 6181.193
  > an1qm3 <- try(nlxb(eunsc, start=start1, data=weeddata1, masked=c("b3")))
  > print(an1qm3) # b3 masked
$resid
 [1]  -5.215005  -6.987727  -8.956014 -11.039413 -12.294541
 [6] -11.440674  -6.030417   5.844013  11.079383   8.211898
[11]  -0.323347 -14.493178

$jacobian
               b1           b2 b3
 [1,] 0.001183578 -4.049119e-05  0
 [2,] 0.003210768 -1.096201e-04  0
 [3,] 0.008679886 -2.947176e-04  0
 [4,] 0.023247649 -7.777528e-04  0
 [5,] 0.060766290 -1.954855e-03  0
 [6,] 0.149563443 -4.356578e-03  0
 [7,] 0.323435267 -7.495055e-03  0
 [8,] 0.565120711 -8.417595e-03  0
 [9,] 0.779365194 -5.889699e-03  0
[10,] 0.905678127 -2.925933e-03  0
[11,] 0.963100894 -1.217211e-03  0
```

```
[12,] 0.986101397 -4.694300e-04  0

$feval
[1] 48

$jeval
[1] 31

$coeffs
[1]   78.57085 2293.94688    1.00000

$ssquares
[1] 1031.011
```

```
> # Note that the parameters are put in out of order to test code.
> an1qm123 <- try(nlxb(eunsc, start=start1, data=weeddata1, masked=c("b2","b1","b3")))
> print(an1qm123) # ALL masked - fails!!
```

```
[1] "Error in nlxb(eunsc, start = start1, data = weeddata1, masked = c(\"b2\",  : \n  All parameters are masked\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in nlxb(eunsc, start = start1, data = weeddata1, masked = c("b2",     "b1", "b3")): All parameters are masked>
```

Finally (for `nlxb`) we combine the bounds and mask.

```
> ## BOUNDS and MASK
> an1qbm2 <- try(nlxb(eunsc, start=startf1, data=weeddata1,
+     lower=c(0,0,0), upper=c(200, 60, .3), masked=c("b2")))
> print(an1qbm2)
```

```
$resid
 [1]  22.387335  22.901372  22.856287  21.850150  19.708919
 [6]  15.468341   8.910727   3.298868  -6.980586 -18.627939
[11] -30.690199 -45.826592

$jacobian
             b1 b2       b3
 [1,] 0.5494911  0 12.47699
 [2,] 0.5980219  0 24.23234
 [3,] 0.6447051  0 34.63517
 [4,] 0.6887881  0 43.21632
 [5,] 0.7296948  0 49.70632
 [6,] 0.7670431  0 54.03716
 [7,] 0.8006408  0 56.31420
 [8,] 0.8304640  0 56.76997
 [9,] 0.8566247  0 55.71260
[10,] 0.8793351  0 53.47875
[11,] 0.8988729  0 50.39696
[12,] 0.9155510  0 46.76318

$feval
[1] 49

$jeval
[1] 27

$coeffs
[1] 50.4017861  1.0000000  0.1986149

$ssquares
[1] 6181.193
```

```
> an1qbm2x <- try(nlxb(eunsc, start=startf1, data=weeddata1,
+     lower=c(0,0,0), upper=c(48, 60, .3), masked=c("b2")))
> print(an1qbm2x)
```

```
$resid
 [1]  21.273604  21.864149  21.875861  20.900559  18.761311
 [6]  14.494232   7.884623   2.199992  -8.167494 -19.912593
[11] -32.077285 -47.316594

$jacobian
      b1 b2       b3
 [1,]  0  0 11.86115
 [2,]  0  0 22.91449
 [3,]  0  0 32.47137
 [4,]  0  0 40.05119
 [5,]  0  0 45.42122
 [6,]  0  0 48.58588
 [7,]  0  0 49.73849
 [8,]  0  0 49.19495
 [9,]  0  0 47.32756
[10,]  0  0 44.51110
[11,]  0  0 41.08606
[12,]  0  0 37.33855

$feval
[1] 37

$jeval
[1] 19

$coeffs
[1] 48.0000000  1.0000000  0.2159692

$ssquares
[1] 6206.102
```

Turning to the function-based `nlfb`,

```
> hobbs.res <- function(x){ # Hobbs weeds problem -- residual
+     if(length(x) != 3) stop("hobbs.res -- parameter vector n!=3")
+     y <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948,
+          75.995, 91.972)
+     tt <- 1:12
+     res <- x[1]/(1+x[2]*exp(-x[3]*tt)) - y
+ }
> hobbs.jac <- function(x) { # Hobbs weeds problem -- Jacobian
+     jj <- matrix(0.0, 12, 3)
+     tt <- 1:12
+     yy <- exp(-x[3]*tt)
+     zz <- 1.0/(1+x[2]*yy)
+     jj[tt,1]  <-  zz
+     jj[tt,2]  <-  -x[1]*zz*zz*yy
+     jj[tt,3]  <-  x[1]*zz*zz*yy*x[2]*tt
+     return(jj)
+ }
> # Check unconstrained
> ans1 <- nlfb(start1, hobbs.res, hobbs.jac)
> ans1
$resid
 [1]  0.01189993 -0.03275547  0.09202996  0.20878182
 [5]  0.39263404 -0.05759436 -1.10572842  0.71578576
```

```
 [9] -0.10764762 -0.34839635  0.65259251 -0.28756791

$jacobian
           [,1]       [,2]       [,3]
 [1,] 0.02711658 -0.1054282   5.175642
 [2,] 0.03673674 -0.1414187  13.884948
 [3,] 0.04959588 -0.1883714  27.742382
 [4,] 0.06664474 -0.2485844  48.813666
 [5,] 0.08900539 -0.3240359  79.537273
 [6,] 0.11792062 -0.4156794 122.438293
 [7,] 0.15463505 -0.5224121 179.522463
 [8,] 0.20018622 -0.6398588 251.293721
 [9,] 0.25510631 -0.7594104 335.526319
[10,] 0.31908250 -0.8682775 426.251654
[11,] 0.39068787 -0.9513292 513.725387
[12,] 0.46733360 -0.9948174 586.046596

$feval
[1] 37

$jeval
[1] 24

$coeffs
[1] 196.1862618  49.0916395   0.3135697

$ssquares
[1] 2.587277
```

```
> ## No jacobian - use internal approximation
> ans1n <- nlfb(start1, hobbs.res)
> ans1n
```

```
$resid
 [1]  0.01189991 -0.03275549  0.09202994  0.20878180
 [5]  0.39263403 -0.05759436 -1.10572841  0.71578577
 [9] -0.10764760 -0.34839633  0.65259252 -0.28756794

$jacobian
           [,1]       [,2]       [,3]
 [1,] 0.02711658 -0.1054282   5.175643
 [2,] 0.03673674 -0.1414186  13.884948
 [3,] 0.04959588 -0.1883714  27.742383
 [4,] 0.06664474 -0.2485844  48.813668
 [5,] 0.08900539 -0.3240359  79.537278
 [6,] 0.11792062 -0.4156793 122.438302
 [7,] 0.15463505 -0.5224121 179.522477
 [8,] 0.20018622 -0.6398587 251.293740
 [9,] 0.25510631 -0.7594104 335.526342
[10,] 0.31908251 -0.8682774 426.251677
[11,] 0.39068787 -0.9513291 513.725406
[12,] 0.46733360 -0.9948173 586.046600

$feval
[1] 40

$jeval
[1] 22

$coeffs
[1] 196.1862605  49.0916393   0.3135697

$ssquares
[1] 2.587277
```

```
> # Bounds -- infeasible start
> ans2i <- try(nlfb(start1, hobbs.res, hobbs.jac,
+    lower=c(b1=0, b2=0, b3=0), upper=c(b1=500, b2=100, b3=.25)))
```

```
  > ans2i
[1] "Error in nlfb(start1, hobbs.res, hobbs.jac, lower = c(b1 = 0, b2 = 0,  : \n  Infeasible start\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in nlfb(start1, hobbs.res, hobbs.jac, lower = c(b1 = 0, b2 = 0,     b3 = 0), upper = c(b1 = 500, b2 = 100, b3 = 0.25
  > # Bounds -- feasible start
  > ans2f <- nlfb(startf1, hobbs.res, hobbs.jac,
  +    lower=c(b1=0, b2=0, b3=0), upper=c(b1=500, b2=100, b3=.25))
  > ans2f
$resid
 [1]  1.8873141  1.9613574  2.1153439  2.1255016  2.0179261
 [6]  1.0532232 -0.7344513  0.1965226 -1.4660939 -2.1115581
[11] -0.4888194  0.9924830

$jacobian
      [,1]       [,2] [,3]
 [1,]    0 -0.0806410    0
 [2,]    0 -0.1027038    0
 [3,]    0 -0.1305065    0
 [4,]    0 -0.1653582    0
 [5,]    0 -0.2087536    0
 [6,]    0 -0.2623256    0
 [7,]    0 -0.3277429    0
 [8,]    0 -0.4065236    0
 [9,]    0 -0.4997414    0
[10,]    0 -0.6076051    0
[11,]    0 -0.7289289    0
[12,]    0 -0.8605590    0

$feval
[1] 31

$jeval
[1] 16

$coeffs
[1] 500.00000  87.94248   0.25000

$ssquares
[1] 29.99273
  > # Mask b2
  > ansm2 <- nlfb(start1, hobbs.res, hobbs.jac, maskidx=c(2))
  > ansm2
$resid
 [1]  22.387335  22.901372  22.856287  21.850150  19.708919
 [6]  15.468341   8.910727   3.298868  -6.980586 -18.627939
[11] -30.690199 -45.826592

$jacobian
           [,1] [,2]     [,3]
 [1,] 0.5494911    0 12.47699
 [2,] 0.5980219    0 24.23234
 [3,] 0.6447051    0 34.63517
 [4,] 0.6887881    0 43.21632
 [5,] 0.7296948    0 49.70632
 [6,] 0.7670431    0 54.03717
 [7,] 0.8006408    0 56.31420
 [8,] 0.8304640    0 56.76997
 [9,] 0.8566247    0 55.71260
[10,] 0.8793351    0 53.47875
[11,] 0.8988729    0 50.39696
[12,] 0.9155510    0 46.76318
```

```
$feval
[1] 56

$jeval
[1] 32

$coeffs
[1] 50.4017865  1.0000000  0.1986149

$ssquares
[1] 6181.193
```

```
> # Mask b3
> ansm3 <- nlfb(start1, hobbs.res, hobbs.jac, maskidx=c(3))
> ansm3
```

```
$resid
 [1]  -5.215005  -6.987727  -8.956014 -11.039413 -12.294541
 [6] -11.440674  -6.030417   5.844013  11.079383   8.211898
[11]  -0.323347 -14.493178

$jacobian
            [,1]          [,2] [,3]
 [1,] 0.001183578 -4.049119e-05    0
 [2,] 0.003210768 -1.096201e-04    0
 [3,] 0.008679886 -2.947176e-04    0
 [4,] 0.023247649 -7.777528e-04    0
 [5,] 0.060766290 -1.954855e-03    0
 [6,] 0.149563443 -4.356578e-03    0
 [7,] 0.323435267 -7.495055e-03    0
 [8,] 0.565120711 -8.417595e-03    0
 [9,] 0.779365194 -5.889699e-03    0
[10,] 0.905678127 -2.925933e-03    0
[11,] 0.963100894 -1.217211e-03    0
[12,] 0.986101397 -4.694300e-04    0

$feval
[1] 48

$jeval
[1] 31

$coeffs
[1]   78.57085 2293.94688    1.00000

$ssquares
[1] 1031.011
```

```
> # Mask all -- should fail
> ansma <- try(nlfb(start1, hobbs.res, hobbs.jac, maskidx=c(3,1,2)))
> ansma
```

```
[1] "Error in nlfb(start1, hobbs.res, hobbs.jac, maskidx = c(3, 1, 2)) : \n  All parameters are masked\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in nlfb(start1, hobbs.res, hobbs.jac, maskidx = c(3, 1, 2)): All parameters are masked>
```

```
> # Bounds and mask
> ansmbm2 <- nlfb(startf1, hobbs.res, hobbs.jac, maskidx=c(2),
+        lower=c(0,0,0), upper=c(200, 60, .3))
> ansmbm2
```

```
$resid
 [1]  22.387335  22.901372  22.856287  21.850150  19.708920
 [6]  15.468341   8.910727   3.298868  -6.980586 -18.627939
[11] -30.690198 -45.826592
```

```
$jacobian
          [,1] [,2]     [,3]
 [1,] 0.5494911    0 12.47699
 [2,] 0.5980219    0 24.23234
 [3,] 0.6447051    0 34.63517
 [4,] 0.6887881    0 43.21632
 [5,] 0.7296948    0 49.70632
 [6,] 0.7670431    0 54.03717
 [7,] 0.8006408    0 56.31420
 [8,] 0.8304640    0 56.76997
 [9,] 0.8566247    0 55.71260
[10,] 0.8793351    0 53.47875
[11,] 0.8988729    0 50.39696
[12,] 0.9155510    0 46.76318

$feval
[1] 50

$jeval
[1] 28

$coeffs
[1] 50.4017865  1.0000000  0.1986149

$ssquares
[1] 6181.193
```

```
> # Active bound
> ansmbm2x <- nlfb(startf1, hobbs.res, hobbs.jac, maskidx=c(2),
+       lower=c(0,0,0), upper=c(48, 60, .3))
> ansmbm2x
```

```
$resid
 [1]  21.273603  21.864149  21.875861  20.900559  18.761311
 [6]  14.494231   7.884623   2.199992  -8.167494 -19.912594
[11] -32.077285 -47.316594

$jacobian
      [,1] [,2]     [,3]
 [1,]    0    0 11.86115
 [2,]    0    0 22.91449
 [3,]    0    0 32.47137
 [4,]    0    0 40.05119
 [5,]    0    0 45.42122
 [6,]    0    0 48.58588
 [7,]    0    0 49.73850
 [8,]    0    0 49.19495
 [9,]    0    0 47.32756
[10,]    0    0 44.51110
[11,]    0    0 41.08606
[12,]    0    0 37.33855

$feval
[1] 35

$jeval
[1] 17

$coeffs
[1] 48.0000000  1.0000000  0.2159692

$ssquares
[1] 6206.102
```

The results match those of `nlxb()`

Finally, let us check the results above with `Rvmmin` and `Rcgmin`. Note that this vignette cannot be created on systems that lack these codes.

```
> require(Rcgmin)
> require(Rvmmin)
> hobbs.f <- function(x) {
+    res<-hobbs.res(x)
+    as.numeric(crossprod(res))
+ }
> hobbs.g <- function(x) {
+    res <- hobbs.res(x) # Probably already available
+    JJ <- hobbs.jac(x)
+    2.0*as.numeric(crossprod(JJ, res))
+ }
> # Check unconstrained
> a1cg <- Rcgmin(start1, hobbs.f, hobbs.g)
> a1cg
$par
        b1         b2         b3
196.1843794  49.0908678   0.3135696

$value
[1] 2.587277

$counts
[1] 1004  351

$convergence
[1] 1

$message
[1] "Too many function evaluations (> 1000) "
> a1vm <- Rvmmin(start1, hobbs.f, hobbs.g)
> a1vm
$par
        b1         b2         b3
196.1862624  49.0916395   0.3135697

$value
[1] 2.587277

$counts
[1] 199  52

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 1 1
> ## No jacobian - use internal approximation
> a1cgn <- try(Rcgmin(start1, hobbs.f))
function(x) {
   res<-hobbs.res(x)
   as.numeric(crossprod(res))
}
> a1cgn
$par
        b1         b2         b3
```

```
196.1861913  49.0916219   0.3135698

$value
[1] 2.587277

$counts
[1] 775 258

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"
```

```
> a1vmn <- try(Rvmmin(start1, hobbs.f))
> a1vmn
```

```
$par
        b1         b2         b3
196.1870411  49.0915204   0.3135689

$value
[1] 2.587277

$counts
[1] 139  48

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 1 1
```

```
> # But
> grfwd <- function(par, userfn, fbase=NULL, eps=1.0e-7, ...) {
+    # Forward different gradient approximation
+    if (is.null(fbase)) fbase <- userfn(par, ...)  # ensure we function value at par
+    df <- rep(NA, length(par))
+    teps <- eps * (abs(par) + eps)
+    for (i in 1:length(par)) {
+       dx <- par
+       dx[i] <- dx[i] + teps[i]
+       df[i] <- (userfn(dx, ...) - fbase)/teps[i]
+    }
+    df
+ }
> a1vmn <- try(Rvmmin(start1, hobbs.f, gr="grfwd"))
> a1vmn
```

```
[1] "Error in mygr(bvec, ...) : could not find function \"gr\"\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in mygr(bvec, ...): could not find function "gr">
```

```
> # Bounds -- infeasible start
> # Note: These codes move start to nearest bound
> a1cg2i <- Rcgmin(start1, hobbs.f, hobbs.g,
+     lower=c(b1=0, b2=0, b3=0), upper=c(b1=500, b2=100, b3=.25))
```

```
  > a1cg2i
$par
      b1       b2       b3
500.00000  87.94248   0.25000

$value
[1] 29.99273

$counts
[1] 87 45

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] -1  1 -1
  > a1vm2i <- Rvmmin(start1, hobbs.f, hobbs.g,
  +     lower=c(b1=0, b2=0, b3=0), upper=c(b1=500, b2=100, b3=.25))
  > a1vm2i # Fails to get to solution!
$par
      b1       b2       b3
500.00000  87.94248   0.25000

$value
[1] 29.99273

$counts
[1] 389 137

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 1 1
  > # Bounds -- feasible start
  > a1cg2f <- Rcgmin(startf1, hobbs.f, hobbs.g,
  +     lower=c(b1=0, b2=0, b3=0), upper=c(b1=500, b2=100, b3=.25))
  > a1cg2f
$par
      b1       b2       b3
500.00000  87.94248   0.25000

$value
[1] 29.99273

$counts
[1] 67 34

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] -1  1 -1
```

```
  > a1vm2f <- Rvmmin(startf1, hobbs.f, hobbs.g,
  +     lower=c(b1=0, b2=0, b3=0), upper=c(b1=500, b2=100, b3=.25))
  > a1vm2f # Gets there, but only just!
$par
      b1       b2       b3
499.96460  87.93419   0.25000

$value
[1] 29.99373

$counts
[1] 3001  494

$convergence
[1] 1

$message
[1] "Too many function evaluations"

$bdmsk
[1]  1  1 -1
  > # Mask b2
  > a1cgm2 <- Rcgmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,0,1))
  > a1cgm2
$par
       b1        b2        b3
50.4017867  1.0000000  0.1986149

$value
[1] 6181.193

$counts
[1] 112  39

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] 1 0 1
  > a1vmm2 <- Rvmmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,0,1))
  > a1vmm2
$par
       b1        b2        b3
50.4017867  1.0000000  0.1986149

$value
[1] 6181.193

$counts
[1] 58 14

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 0 1
```

```
  > # Mask b3
  > a1cgm3 <- Rcgmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,1,0))
  > a1cgm3
$par
       b1         b2         b3
  78.57081 2293.93952    1.00000

$value
[1] 1031.011

$counts
[1] 181  80

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] 1 1 0
  > a1vmm3 <- Rvmmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,1,0))
  > a1vmm3
$par
       b1         b2         b3
  78.57085 2293.94690    1.00000

$value
[1] 1031.011

$counts
[1] 102  32

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 1 0
  > # Mask all -- should fail
  > a1cgma <- Rcgmin(start1, hobbs.f, hobbs.g, bdmsk=c(0,0,0))
  > a1cgma
$par
b1 b2 b3
 1  1  1

$value
[1] 23520.58

$counts
[1] 1 1

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] 0 0 0
```

```
  > a1vmma <- Rvmmin(start1, hobbs.f, hobbs.g, bdmsk=c(0,0,0))
  > a1vmma
$par
b1 b2 b3
 1  1  1

$value
[1] 23520.58

$counts
[1] 1 1

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 0 0 0
  > # Bounds and mask
  > ansmbm2 <- nlfb(startf1, hobbs.res, hobbs.jac, maskidx=c(2),
  +       lower=c(0,0,0), upper=c(200, 60, .3))
  > ansmbm2
$resid
 [1]  22.387335  22.901372  22.856287  21.850150  19.708920
 [6]  15.468341   8.910727   3.298868  -6.980586 -18.627939
[11] -30.690198 -45.826592

$jacobian
          [,1] [,2]     [,3]
 [1,] 0.5494911    0 12.47699
 [2,] 0.5980219    0 24.23234
 [3,] 0.6447051    0 34.63517
 [4,] 0.6887881    0 43.21632
 [5,] 0.7296948    0 49.70632
 [6,] 0.7670431    0 54.03717
 [7,] 0.8006408    0 56.31420
 [8,] 0.8304640    0 56.76997
 [9,] 0.8566247    0 55.71260
[10,] 0.8793351    0 53.47875
[11,] 0.8988729    0 50.39696
[12,] 0.9155510    0 46.76318

$feval
[1] 50

$jeval
[1] 28

$coeffs
[1] 50.4017865  1.0000000  0.1986149

$ssquares
[1] 6181.193
  > a1cgbm2 <- Rcgmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,0,1),
  +        lower=c(0,0,0), upper=c(200, 60, .3))
  > a1cgbm2
$par
       b1         b2         b3
50.4017851  1.0000000  0.1986149

$value
```

```
[1] 6181.193

$counts
[1] 76 29

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] 1 0 1
```

```
> a1vmbm2 <- Rvmmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,0,1),
+         lower=c(0,0,0), upper=c(200, 60, .3))
> a1vmbm2
```

```
$par
       b1        b2        b3
50.4017867  1.0000000  0.1986149

$value
[1] 6181.193

$counts
[1] 75 14

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 0 1
```

```
> # Active bound
> a1cgm2x <- Rcgmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,0,1),
+         lower=c(0,0,0), upper=c(48, 60, .3))
> a1cgm2x
```

```
$par
       b1        b2        b3
48.0000000  1.0000000  0.2159692

$value
[1] 6206.102

$counts
[1] 37 14

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$bdmsk
[1] -1  0  1
```

```
> a1vmm2x <- Rvmmin(start1, hobbs.f, hobbs.g, bdmsk=c(1,0,1),
+         lower=c(0,0,0), upper=c(48, 60, .3))
> a1vmm2x
```

```
$par
       b1        b2        b3
48.0000000  1.0000000  0.2159692
```

```
$value
[1] 6206.102

$counts
[1] 127  50

$convergence
[1] 0

$message
[1] "Converged"

$bdmsk
[1] 1 0 1
```

6 Brief example of `minpack.lm`

Recently Kate Mullen provided some capability for the package `minpack.lm` to include bounds constraints. I am particularly happy that this effort is proceeding, as there are significant differences in how `minpack.lm` and `nlmrt` are built and implemented. They can be expected to have different performance characteristics on different problems. A lively dialogue between developers, and the opportunity to compare and check results can only improve the tools.

The examples below are a very quick attempt to show how to run the Ratkowsky-Huet problem with `nls.lm` from `minpack.lm`.

```
  > require(minpack.lm)
  > anlslm <- nls.lm(ones, lower=rep(-1000,4), upper=rep(1000,4), jres, jjac, yield=pastured
  > cat("anlslm from ones\n")
anlslm from ones
  > print(strwrap(anlslm))
 [1] "c(NaN, NaN, NaN, NaN)"
 [2] "c(NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN,"
 [3] "NaN, NaN, NaN, NaN, NaN, NaN)"
 [4] "c(NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN)"
 [5] "4"
 [6] "The cosine of the angle between `fvec' and any column"
 [7] "of the Jacobian is at most `gtol' in absolute value."
 [8] "list(t1 = 3, t2 = 2.3723939879224e-11, t3 ="
 [9] "5.8039519205899e-10, t4 = 1.27525858056086e-09)"
[10] "3"
[11] "c(17533.3402000004, 16864.5616372991, NaN, 0)"
[12] "NaN"
  > anlslmh <- nls.lm(huetstart, lower=rep(-1000,4), upper=rep(1000,4), jres, jjac, yield=pa
  > cat("anlslmh from huetstart\n")
anlslmh from huetstart
  > print(strwrap(anlslmh))
 [1] "c(69.9551973916736, 61.6814877170941,"
 [2] "-9.20891880263443, 2.37781455978467)"
 [3] "c(9, -4.54037977686007, 105.318033221555,"
 [4] "403.043210394647, -4.54037977686007,"
 [5] "3.51002837648689, -39.5314537948583,"
 [6] "-137.559566823766, 105.318033221555,"
 [7] "-39.5314537948583, 1668.11894086464,"
```

```
 [8] "6495.67702199832, 403.043210394647,"
 [9] "-137.559566823766, 6495.67702199832,"
[10] "25481.4530263827)"
[11] "c(0.480682793156298, 0.669303022602289,"
[12] "-2.28431914156848, 0.84375480165378,"
[13] "0.734587578832198, 0.0665510313004845,"
[14] "-0.985814877917491, -0.0250630130722556,"
[15] "0.500317790294616)"
[16] "1"
[17] "Relative error in the sum of squares is at most"
[18] "`ftol'."
[19] "list(t1 = 3, t2 = 2.35105755434962, t3 ="
[20] "231.250186433367, t4 = 834.778914353853)"
[21] "42"
[22] "c(13386.9099465603, 13365.3097414383,"
[23] "13351.1970260154, 13321.6478455192, 13260.1135652244,"
[24] "13133.6391318145, 12877.8542053848, 12373.5432344283,"
[25] "11428.8257706578, 9832.87890178625, 7138.12187613238,"
[26] "3904.51162830831, 2286.64875980737, 1978.18149980306,"
[27] "1620.89081508973, 1140.58638304326, 775.173148616759,"
[28] "635.256627921485, 383.73614705125, 309.34124999335,"
[29] "219.735856060243, 177.39873817915, 156.718991828473,"
[30] "135.513594568191, 93.4016394568244, 72.8219383036213,"
[31] "66.331560983492, 56.2809616213412, 54.9453021619837,"
[32] "53.6227655715772, 51.9760950696957, 50.1418078879664,"
[33] "48.130702164752, 44.7097757109316, 42.8838792615125,"
[34] "32.3474231559281, 26.5253835687528, 15.3528215541113,"
[35] "14.7215507012991, 8.37980617628204, 8.37589765770224,"
[36] "8.37588365348112, 8.37588355972579)"
[37] "8.37588355972579"
```

References

Elzhov, T.˜V., K.˜M. Mullen, A.-N. Spiess, and B.˜Bolker (2012). *minpack.lm: R interface to the Levenberg-Marquardt nonlinear least-squares algorithm found in MINPACK, plus support for bounds*. R Project for Statistical Computing. R package version 1.1-6.

Huet, S.˜S. et˜al. (1996). *Statistical tools for nonlinear regression: a practical guide with S-PLUS examples*. Springer series in statistics.

Moré, J.˜J., B.˜S. Garbow, and K.˜E. Hillstrom (1980). ANL-80-74, User Guide for MINPACK-1. Technical report.

Nash, J.˜C. (1979). *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Bristol: Adam Hilger. Second Edition, 1990, Bristol: Institute of Physics Publications.

Ratkowsky, D.˜A. (1983). *Nonlinear Regression Modeling: A Unified Practical Approach*. New York and Basel: Marcel Dekker Inc.