Running funconstrain tests in package optimx

John C. Nash (profjcnash *at* gmail.com)

2024-04-08

Abstract

The `funconstrain` package (https://github.com/jlmelville/funconstrain) provides R users with a convenient tool to access the test functions of Moré, Garbow, and Hillstrom (1981). This vignette article describes a program to apply these test functions to solvers in the `optimx` package (Nash and Varadhan (2011)).

Background

Numerical optimization of functions of several, namely `n`, parameters is an important computational task. R (R Development Core Team (2008)) is a major platform for scientific and statistical calculations and has provided tools for numerical optimization and nonlinear least squares since its inception. These have been extended via a number of packages. In particular, the author has been heavily involved in this effort, and in collaboration with others has provided the packate `optimx` which wraps a number of solvers to allow their invocation by a common calling syntax. Note that *optimization* in R generally means *function minimization*, possibly with bounds (or box) constraints on the function parameters.

It is extremely helpful to users to have examples and tests of function minimization. In many situations it is extremely easy to insert an error into code, so easy-to-apply tests allow for the discovery of such errors. There are a number of collections of test functions with many overlaps and minor differences. A well-established and well-documented set of such functions are those of Moré, Garbow, and Hillstrom (1981). These have been translated into R by James Melville in the R package `funconstrain` (https://github.com/jlmelville/funconstrain). While initially these provided the function and its gradient given a set of suitable input parameters, the present author added code to compute the Hessian for each test function. This allows Newton-like solvers to be applied. `funconstrain` also provides suggested initial parameter vectors for each of the 35 test functions. However, where there are multiple input possibilities, just one is provided, for example when the test function has a variable number of parameters.

What is then missing is the link between `funconstrain` and the tools in `optimx`, which this article aims to provide.

Function fufn()

Most of the test functions in (**More1981TU?**) are sums of squares of nonlinear functions. While `n` is the number of parameters, we may have a different number of functions squared in the summation. Call this `m`. This may be altered to give different variations of a given function, so `m` must be provided.

Many of the solvers in `optimx` are capable of handling bounds constraints on the `n` parameters. That is parameter `i` must satisfy

```
lower[i] <= prm[i] <= upper[i]
```

where `prm` is the parameter vector and `lower` and `upper` are vectors of numbers providing lower and upper bounds. Methods in `optimx` that can handle masks are listed in the character vector `bdmeth` returned by the function `optimx::ctrldefault(n)`. Note that a number of parameters `n` must nominally be provided to `ctrldefault()` but generally `n` can be specified as 2 to get the default settings for 'optimx. At time of writing

```
bdmeth <- c("L-BFGS-B", "nlminb", "lbfgsb3c", "Rcgmin", "Rtnmin", "nvm",
"Rvmmin", "bobyqa", "nmkb", "hjkb", "hjn", "snewtonm", "ncg",
"slsqp", "tnewt", "nlnm", "snewtm", "spg")
```

If the upper and lower bound for a parameter are equal, we can say the parameter is **fixed** or **masked**. This may seem to be a silly option, since it essentially reduces the dimensionality of the problem. However, there are many situations where we have evidence that a parameter takes a particular (fixed) value, but know that we may wish to allow optimization over that parameter in later investigations. Masks allow us to avoid having to rewrite the function, gradient and Hessian code. However, only a few optimization solvers handle masks. The function `optimx::ctrldefault()` returns a value `maskmeth` with a list of solvers that do handle the situation where lower and upper bounds coincide. At the time of writing this is specified as

```
maskmeth <- c("Rcgmin", "nvm", "hjn", "ncg", "snewtonm", "nlminb", "L-BFGS-B")
```

With the above in mind, the function `fufn()` was written to access the test functions of `funconstrain`.

The fufn.R code

```
fufn <- function(n=NULL, fnum=NULL){
  # return list with tfn=function, tgr=gradient given fn number and n
  if (is.null(fnum)) stop("ffn needs a function number fnum")
  if ((fnum < 1) || (fnum > 35)) stop("fnum must be in [1, 35]")
#  cat("entering ffn, fnum=",fnum,"\n")
  # select function
  funnam <- c("rosen", "freud_roth", "powell_bs", "brown_bs", "beale",
              "jenn_samp", "helical", "bard", "gauss", "meyer", "gulf",
              "box_3d", "powell_s", "wood", "kow_osb", "brown_den",
              "osborne_1", "biggs_exp6", "osborne_2", "watson", "ex_rosen",
              "ex_powell", "penalty_1", "penalty_2", "var_dim", "trigon",
              "brown_al", "disc_bv", "disc_ie", "broyden_tri", "broyden_band",
              "linfun_fr", "linfun_r1", "linfun_r1z", "chebyquad")
#  print(str(funnam))
  fname <- funnam[as.integer(fnum)]
#  cat("fname:", fname,"\n")
  while (fnum %in% 1:35) {
    ameth <- optimx::ctrldefault(2)$bdmeth # Choose only bounded methods
    ameth <- ameth[ameth != "lbfgsb3c"] ## ?? Temporarily remove lbfgsb3c
    ameth <- c(ameth, "L-BFGS-B")
    # ?? may want to test allmeth to check that inappropriate methods are captured
    #    cat("in while, fnum=", fnum); tmp <- readline("cont.")
    mm <- 0 # in case m value needed
  if (fnum == 1) {
     n <- 2 # fixed
     mm <- 2
     tt <- rosen()
     if (is.function(tt$x0)) {
       xx0<-tt$x0(n)
     }
     else xx0 <- tt$x0
     lo <- rep((min(xx0)-0.1), n)
     up <- rep((max(xx0)+0.1), n)
     break }

  if (fnum == 2) {
     n <- 2 # fixed
```

```
    mm <- 2
    tt <- freud_roth()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 3) {
    n <- 2 # fixed
    mm <- 2
    tt <- powell_bs()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 4) {
    n <- 2 # fixed
    mm <- 3
    tt <- brown_bs()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
## BAD -- reset 20240323
#     lo <- rep((min(xx0)-0.1), n)
#     up <- rep((max(xx0)+0.1), n)
    lo <- -1e20
    up <- -lo
    break }

  if (fnum == 5) {
    n <- 2 # fixed
    mm <- 3
    tt <- beale()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 6) {
    n <- 2 # fixed
    mm <- 10
    tt <- jenn_samp()
```

```r
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 7) {
    n <- 3 # fixed
    tt <- helical()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 8) {
    n <- 3 # fixed
    mm <- 15
    tt <- bard()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 9) {
    n <- 3 # fixed
    mm <- 15
    tt <- gauss()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 10) {
    n <- 3 # fixed
    m <- 16 # ?? how to return
    tt <- meyer()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
```

```
   break }

if (fnum == 11) {
   n <- 3
   mm <- 99
   tt <- gulf()
   if (is.function(tt$x0)) {
     xx0<-tt$x0(n)
   }
   else xx0 <- tt$x0
   lo <- rep((min(xx0)-0.1), n)
   up <- rep((max(xx0)+0.1), n)
   break }

if (fnum == 12) {
   n <- 3
   mm <- 20
   tt <- box_3d()
   if (is.function(tt$x0)) {
     xx0<-tt$x0(n)
   }
   else xx0 <- tt$x0
   lo <- rep((min(xx0)-0.1), n)
   up <- rep((max(xx0)+0.1), n)
   break }

if (fnum == 13) {
   n <- 4
   tt <- powell_s()
   if (is.function(tt$x0)) {
     xx0<-tt$x0(n)
   }
   else xx0 <- tt$x0
   lo <- rep((min(xx0)-0.1), n)
   up <- rep((max(xx0)+0.1), n)
   break }

if (fnum == 14) {
   n <- 4
   tt <- wood()
   if (is.function(tt$x0)) {
     xx0<-tt$x0(n)
   }
   else xx0 <- tt$x0
   lo <- rep((min(xx0)-0.1), n)
   up <- rep((max(xx0)+0.1), n)
   break }

if (fnum == 15) {
   mm <- 11
   n <- 4
   tt <- kow_osb()
   if (is.function(tt$x0)) {
```

```r
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 16) {
    mm <- 20
    n <- 4
    tt <- brown_den()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 17) {
    mm <- 33
    n <- 5
    tt <- osborne_1()
    ameth<-ameth[-which(ameth=="L-BFGS-B")] # remove L-BFGS-B from this case
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    lo[4] <- 0
    lo[5] <- 0
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 18) {
    mm <- 20
    n <- 6
    tt <- biggs_exp6()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 19) {
    mm <- 65
    n <- 11
    tt <- osborne_2()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
```

```
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 20) {
    n <-8
    mm <- 31
    tt <- watson()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 21) {
    n <- 10
    tt <- ex_rosen()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 22) {
    n <- 20
    tt <- ex_powell()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 23) {
    n <- 10
    mm <- n + 1
    tt <- penalty_1()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 24) {
    n <- 10
```

```
    mm <- n + 1
    tt <- penalty_2()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 25) {
    n <- 6
    mm <- n + 2
    tt <- var_dim()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 26) {
    n <- 8
    tt <- trigon()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 27) {
    n <- 8
    mm <- n
    tt <- brown_al()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 28) {
    n <- 6
    mm <- n
    tt <- disc_bv()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
```

```
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

 if (fnum == 29) {
    n <- 8
    mm <- n
    tt <- disc_ie()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

 if (fnum == 30) {
    n <- 8
    mm <- n
    tt <- broyden_tri()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

 if (fnum == 31) {
    n <- 8
    mm <- n
    tt <- broyden_band()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

 if (fnum == 32) {
    mm <- 10
    n <- 8
    tt <- linfun_fr()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

 if (fnum == 33) {
```

```
    mm <- 10
    n <- 8
    tt <- linfun_r1()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 34) {
    mm <- 10
    n <- 8
    tt <- linfun_r1z()
    if (is.function(tt$x0)) {
      xx0<-tt$x0(n)
    }
    else xx0 <- tt$x0
    lo <- rep((min(xx0)-0.1), n)
    up <- rep((max(xx0)+0.1), n)
    break }

  if (fnum == 35) {
     n <- 8
     m <- n
     tt <- chebyquad()
     if (is.function(tt$x0)) {
       xx0<-tt$x0(n)
     }
     else xx0 <- tt$x0
     lo <- rep((min(xx0)-0.1), n)
     up <- rep((max(xx0)+0.1), n)
     break }
  }
# NOTE: bounds are experimental only
  mask <- rep(1L, n) # masks set to "free" (not masked)
  val <- list(npar = n, fffn=tt$fn, ffgr=tt$gr, x0=xx0, lo=lo, up=up,
              mask=mask, fname=fname, ameth=ameth)
#  cat("val:"); print(val); tmp<-readline('exit ffn')
  val
} # end fufn
```

Calling fufn()

While we can write our own driver for `fufn()`, I wanted to make the task extremely easy. Thus the script `fufnrun.R` is provided. This is set up to use a simple text file to specify which test functions are to be applied to which solvers. Moreover, a “sink” function name can be specified to save the text output of the run.

Test specification file RFO.txt

Let us consider an example.

```
testsink230324A.txt
1, 6:8, 35
c("L-BFGS-B",  "lbfgsb3c", "lbfgs", "lbfgsb3")
FALSE
```

The lines of the above file provide the following information:

- the first line is the name of the text file to use to save the output
- line 2 says that test functions 1, 6, 7, 8, and 35 are to be used. Note that we can use the colon ":" when giving a contiguous range of function numbers. These numbers – by referring back to the vector `funnam` at the top of function `fufn()` – specify functions "rosen", "jenn-samp", "helical", "bard" and "chebyquad". Using the function numbers. Appendix A lists the numbers and corresponding names.
- line 3 gives an R character vector of the solver methods to be applied.

' line 4 is TRUE if the experimental bounds constraints are to be applied.

Appendix A: function numbers and names

```
1       rosen
2       freud_roth
3       powell_bs
4       brown_bs
5       beale
6       jenn_samp
7       helical
8       bard
9       gauss
10      meyer
11      gulf
12      box_3d
13      powell_s
14      wood
15      kow_osb
16      brown_den
17      osborne_1
18      biggs_exp6
19      osborne_2
20      watson
21      ex_rosen
22      ex_powell
23      penalty_1
24      penalty_2
25      var_dim
26      trigon
27      brown_al
28      disc_bv
29      disc_ie
30      broyden_tri
31      broyden_band
32      linfun_fr
33      linfun_r1
34      linfun_r1z
35      chebyquad
```

References

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

Nash, John C, and Ravi Varadhan. 2011. Optimx*: A Replacement and Extension of the optim() Function*. Nash Information Services Inc.; Johns Hopkins University.

R Development Core Team. 2008. ***R****: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org.