Rvmmin - an R implementation of the Fletcher(1970) variable metric method with bounds and masks

John Nash

2017-07-18

Rvmmin description, examples and tests

Rvmmin is an all-R version of the Fletcher-Nash variable metric nonlinear parameter optimization code of (**Fletcher70?**) as modified by Nash (1979).

This vignette is intended to show various features of the package, so it is rather detailed and "busy". However, it is also hopefully helpful in showing how to use the method for more difficult problems. Note that as of 2023-6-21, the function `nvm()` is being developed to work under the `optimr()` function as a more compact alternative.

Algorithm implementation

Fletcher's variable metric method attempts to mimic Newton's iteration for function minimization approximately.

Newton's method starts with an original set of parameters x_0. At a given iteraion, which could be the first, we want to solve

$$x_{k+1} = x_k - H^{-1} g$$

where H is the Hessian and g is the gradient at x_k.

Newton's method is unattractive in general function minimization situations because

- evaluating the Hessian is generally time consuming and error prone;
- solving the equation

  $$Hdelta = -g$$

  (which is much less computational effort than inverting H), is still a lot of work which needs to be carried out every iteration.

While the base Newton algorithm is as given, generally we carry out some sort of line search along the search direction delta from the current iterate x_k. Indeed, many otherwise highly educated workers try to implement it without paying attention to safeguarding the iterations and ensuring appropriate progress towards a minimum.

Termination nuances

Termination variation with control tolerances

Let us use the Chebyquad test problem in n=4 parameters with different controls **eps** and **acctol** and tabulate the results to explore how our results change with different values of these program control inputs.

```
cyq.f <- function (x) {
  rv<-cyq.res(x)
  f<-sum(rv*rv)
}

cyq.res <- function (x) {
# Fletcher's chebyquad function m = n -- residuals
   n<-length(x)
   res<-rep(0,n) # initialize
   for (i in 1:n) { #loop over resids
     rr<-0.0
     for (k in 1:n) {
  z7<-1.0
  z2<-2.0*x[k]-1.0
        z8<-z2
        j<-1
        while (j<i) {
            z6<-z7
            z7<-z8
            z8<-2*z2*z7-z6 # recurrence to compute Chebyshev polynomial
            j<-j+1
        } # end recurrence loop
        rr<-rr+z8
      } # end loop on k
      rr<-rr/n
      if (2*trunc(i/2) == i) { rr <- rr + 1.0/(i*i - 1) }
      res[i]<-rr
    } # end loop on i
    res
}

cyq.jac<- function (x) {
#  Chebyquad Jacobian matrix
   n<-length(x)
   cj<-matrix(0.0, n, n)
   for (i in 1:n) { # loop over rows
     for (k in 1:n) { # loop over columns (parameters)
       z5<-0.0
       cj[i,k]<-2.0
       z8<-2.0*x[k]-1.0
       z2<-z8
       z7<-1.0
       j<- 1
       while (j<i) { # recurrence loop
         z4<-z5
         z5<-cj[i,k]
         cj[i,k]<-4.0*z8+2.0*z2*z5-z4
         z6<-z7
         z7<-z8
         z8<-2.0*z2*z7-z6
         j<- j+1
       } # end recurrence loop
       cj[i,k]<-cj[i,k]/n
```

```
    } # end loop on k
  } # end loop on i
  cj
}


cyq.g <- function (x) {
  cj<-cyq.jac(x)
  rv<-cyq.res(x)
  gg<- as.vector(2.0* rv %*% cj)
}

require(optimx)
```

```
## Loading required package: optimx
```

```
nn <- 4
xx0 <- 1:nn
xx0 <- xx0 / (nn+1.0) # Initial value suggested by Fletcher

# cat("aed\n")
# aed <- Rvmminu(xx0, cyq.f, cyq.g, control=list(trace=2, checkgrad=FALSE))
# print(aed)
#================================
# Now build a table of results for different values of eps and acc
veps <- c(1e-3, 1e-5, 1e-7, 1e-9, 1e-11)
vacc <- c(.1, .01, .001, .0001, .00001, .000001)
resdf <- data.frame(eps=NA, acctol=NA, nf=NA, ng=NA, fval=NA, gnorm=NA)
for (eps in veps) {
  for (acctol in vacc) {
    ans <- Rvmminu(xx0, cyq.f, cyq.g,
          control=list(eps=eps, acctol=acctol, trace=0))
    gn <- as.numeric(crossprod(cyq.g(ans$par)))
    resdf <- rbind(resdf,
              c(eps, acctol, ans$counts[1], ans$counts[2], ans$value, gn))
  }
}
resdf <- resdf[-1,]
# Display the function value found for different tolerances
xtabs(formula = fval ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol         1e-11        1e-09        1e-07        1e-05        0.001
##   1e-06 3.964816e-29 3.964816e-29 3.964816e-29 6.841589e-24 7.489034e-15
##   1e-05 3.964816e-29 3.964816e-29 3.964816e-29 6.841589e-24 7.489034e-15
##   1e-04 3.964816e-29 3.964816e-29 3.964816e-29 6.841589e-24 7.489034e-15
##   0.001 3.964816e-29 3.964816e-29 3.964816e-29 6.841589e-24 7.489034e-15
##   0.01  3.964816e-29 3.964816e-29 3.964816e-29 6.841589e-24 7.489034e-15
##   0.1   3.964816e-29 3.964816e-29 3.964816e-29 6.841589e-24 7.489034e-15
```

```
# Display the gradient norm found for different tolerances
xtabs(formula = gnorm ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol         1e-11        1e-09        1e-07        1e-05        0.001
```

```
##   1e-06 2.130888e-27 2.130888e-27 2.130888e-27 3.528814e-22 1.091331e-13
##   1e-05 2.130888e-27 2.130888e-27 2.130888e-27 3.528814e-22 1.091331e-13
##   1e-04 2.130888e-27 2.130888e-27 2.130888e-27 3.528814e-22 1.091331e-13
##   0.001 2.130888e-27 2.130888e-27 2.130888e-27 3.528814e-22 1.091331e-13
##   0.01  2.130888e-27 2.130888e-27 2.130888e-27 3.528814e-22 1.091331e-13
##   0.1   2.130888e-27 2.130888e-27 2.130888e-27 3.528814e-22 1.091331e-13
```

```
# Display the number of function evaluations used for different tolerances
xtabs(formula = nf ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol  1e-11 1e-09 1e-07 1e-05 0.001
##   1e-06    20    20    20    17    12
##   1e-05    20    20    20    17    12
##   1e-04    20    20    20    17    12
##   0.001    20    20    20    17    12
##   0.01     20    20    20    17    12
##   0.1      20    20    20    17    12
```

```
# Display the number of gradient evaluations used for different tolerances
xtabs(formula = ng ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol  1e-11 1e-09 1e-07 1e-05 0.001
##   1e-06    15    15    15    12     9
##   1e-05    15    15    15    12     9
##   1e-04    15    15    15    12     9
##   0.001    15    15    15    12     9
##   0.01     15    15    15    12     9
##   0.1      15    15    15    12     9
```

Here – and we caution that this is but a single instance of a single test problem – the differences in results and level of effort to obtain them are regulated by the values of `eps` only. This control is used to judge the size of the gradient norm and the gradient projection on the search vector.

Problems of function scale

One of the more difficult aspects of termination decisions is that we need to decide when we have a "nearly" zero gradient. However, this "zero gradient" is relative to the overall scale of the function. Let us see what happens when we consider solving a problem where the function scale is adjustable. Note that we multiply the constant sequence `yy` by `pi/4` to avoid integer values which may give results that are fortuitously better than may be normally found.

```
sq<-function(x, exfs=1){
  nn<-length(x)
  yy<-(1:nn)*pi/4
  f<-(10^exfs)*sum((yy-x)^2)
  f
}
sq.g <- function(x, exfs=1){
  nn<-length(x)
  yy<-(1:nn)*pi/4
  gg<- 2*(x - yy)*(10^exfs)
}
require(optimx)
nn <- 4
xx0 <- rep(pi, nn) # crude start
```

```
# Now build a table of results for different values of eps and acc
veps <- c(1e-3, 1e-5, 1e-7, 1e-9, 1e-11)
exfsi <- 1:6
resdf <- data.frame(eps=NA, exfs=NA, nf=NA, ng=NA, fval=NA, gnorm=NA)
for (eps in veps) {
  for (exfs in exfsi) {
    ans <- Rvmminu(xx0, sq, sq.g,
                   control=list(eps=eps, trace=0), exfs=exfs)
    gn <- as.numeric(crossprod(sq.g(ans$par)))
    resdf <- rbind(resdf,
                   c(eps, exfs, ans$counts[1], ans$counts[2], ans$value, gn))
  }
}
resdf <- resdf[-1,]
# Display the function value found for different tolerances
xtabs(formula = fval ~ exfs + eps, data=resdf)
```

```
##     eps
## exfs        1e-11        1e-09        1e-07        1e-05        0.001
##    1 2.576124e-29 2.576124e-29 2.576124e-29 1.903127e-28 1.903127e-28
##    2 0.000000e+00 0.000000e+00 0.000000e+00 1.425669e-25 1.425669e-25
##    3 1.232595e-29 1.232595e-29 1.232595e-29 1.232595e-29 1.178643e-22
##    4 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 4.149267e-20
##    5 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 3.620953e-20
##    6 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
```

```
# Display the gradient norm found for different tolerances
xtabs(formula = gnorm ~ exfs + eps, data=resdf)
```

```
##     eps
## exfs        1e-11        1e-09        1e-07        1e-05        0.001
##    1 1.030450e-27 1.030450e-27 1.030450e-27 7.612508e-27 7.612508e-27
##    2 0.000000e+00 0.000000e+00 0.000000e+00 5.702675e-25 5.702675e-25
##    3 4.930381e-30 4.930381e-30 4.930381e-30 4.930381e-30 4.714574e-23
##    4 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 1.659707e-21
##    5 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 1.448381e-22
##    6 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
```

```
# Display the number of function evaluations used for different tolerances
xtabs(formula = nf ~ exfs + eps, data=resdf)
```

```
##     eps
## exfs 1e-11 1e-09 1e-07 1e-05 0.001
##    1    11    11    11     5     5
##    2    16    16    16     6     6
##    3    21    21    21    21     8
##    4    25    25    25    25     9
##    5    31    31    31    31    21
##    6    35    35    35    35    35
```

```
# Display the number of gradient evaluations used for different tolerances
xtabs(formula = ng ~ exfs + eps, data=resdf)
```

```
##     eps
## exfs 1e-11 1e-09 1e-07 1e-05 0.001
##    1     5     5     5     3     3
```

```
##    2     7     7     7     3     3
##    3     7     7     7     7     3
##    4     7     7     7     7     3
##    5     7     7     7     7     5
##    6     7     7     7     7     7
```

The general tendency here is for the amount of work in terms of function evaluations to rise with the function scale and with tighter (smaller) test tolerances, while the quality of the solution is poorer with larger scale and also larger (looser) tolerances. However, some exceptions can be seen, though the overall quality of solutions (function and gradient norm) is very good. Moreover, the number of gradient evaluations does not climb notably with the scale or inverse tolerance.

Problems of parameter scale

There are similar issues of parameter scaling. Let us look at very simple sum of squares function where we scale the parameters in a nasty way.

```
ssq.f<-function(x){
   nn<-length(x)
   yy <- 1:nn
   f<-sum((yy-x/10^yy)^2)
   f
}
ssq.g <- function(x){
   nn<-length(x)
   yy<-1:nn
   gg<- 2*(x/10^yy - yy)*(1/10^yy)
}

xy <- c(1, 1/10, 1/100, 1/1000)
# note: gradient was checked using numDeriv
veps <- c(1e-3, 1e-5, 1e-7, 1e-9, 1e-11)
vacc <- c(.1, .01, .001, .0001, .00001, .000001)
resdf <- data.frame(eps=NA, acctol=NA, nf=NA, ng=NA, fval=NA, gnorm=NA)
for (eps in veps) {
  for (acctol in vacc) {
    ans <- Rvmminu(xy, ssq.f, ssq.g,
          control=list(eps=eps, acctol=acctol, trace=0))
    gn <- as.numeric(crossprod(ssq.g(ans$par)))
    resdf <- rbind(resdf,
               c(eps, acctol, ans$counts[1], ans$counts[2], ans$value, gn))
  }
}
resdf <- resdf[-1,]
# Display the function value found for different tolerances
xtabs(formula = fval ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol         1e-11        1e-09        1e-07        1e-05        0.001
##   1e-06 0.000000e+00 0.000000e+00 1.475416e-29 5.767419e-19 8.977439e-11
##   1e-05 0.000000e+00 0.000000e+00 1.475416e-29 5.767419e-19 8.977439e-11
##   1e-04 0.000000e+00 0.000000e+00 1.475416e-29 5.767419e-19 8.977439e-11
##   0.001 0.000000e+00 0.000000e+00 1.475416e-29 5.767419e-19 8.977439e-11
##   0.01  0.000000e+00 0.000000e+00 1.475416e-29 5.767419e-19 8.977439e-11
##   0.1   0.000000e+00 0.000000e+00 1.475416e-29 5.767419e-19 8.977439e-11
```

```
# Display the gradient norm found for different tolerances
xtabs(formula = gnorm ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol         1e-11        1e-09        1e-07        1e-05        0.001
##   1e-06 0.000000e+00 0.000000e+00 7.783028e-33 3.430257e-23 3.473135e-14
##   1e-05 0.000000e+00 0.000000e+00 7.783028e-33 3.430257e-23 3.473135e-14
##   1e-04 0.000000e+00 0.000000e+00 7.783028e-33 3.430257e-23 3.473135e-14
##   0.001 0.000000e+00 0.000000e+00 7.783028e-33 3.430257e-23 3.473135e-14
##   0.01  0.000000e+00 0.000000e+00 7.783028e-33 3.430257e-23 3.473135e-14
##   0.1   0.000000e+00 0.000000e+00 7.783028e-33 3.430257e-23 3.473135e-14
```

```
# Display the number of function evaluations used for different tolerances
xtabs(formula = nf ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol  1e-11 1e-09 1e-07 1e-05 0.001
##   1e-06    56    56    55    53    51
##   1e-05    56    56    55    53    51
##   1e-04    56    56    55    53    51
##   0.001    56    56    55    53    51
##   0.01     56    56    55    53    51
##   0.1      56    56    55    53    51
```

```
# Display the number of gradient evaluations used for different tolerances
xtabs(formula = ng ~ acctol + eps, data=resdf)
```

```
##        eps
## acctol  1e-11 1e-09 1e-07 1e-05 0.001
##   1e-06    56    56    55    53    51
##   1e-05    56    56    55    53    51
##   1e-04    56    56    55    53    51
##   0.001    56    56    55    53    51
##   0.01     56    56    55    53    51
##   0.1      56    56    55    53    51
```

The results above suggest that parameter scaling is not much of a problem. Actually, these are the very best results I have found with any method for this problem, which is actually rather nasty. I suggest trying this problem on your favourite optimizer. Alternatively, use the package `optimr` and run the function `opm()` with `method="ALL"`.

Weeds problem with random starts

This notorious problem (see Nash (1979), page 120, Nash (2014), page 205, for details under the heading **Hobbs Weeds problem**) is small but generally difficult due to the possibility of bad scaling of both function and parameters and a near-singular Hessian in the original parameterization.

The Fletcher variable metric method can solve this problem quite well, though default termination settings should be overridden. It is important to ensure there are enough iterations to allow the method to "grind" at the problem. If one uses default settings for maxit in optim:BFGS, then the success rate drops to less than 2/3 of cases tried below.

Below we use 100 "random" starting points for both Rvmmin and the optim:BFGS minimizers (which should be, but are not quite, the same).

```
## hobbstarts.R -- starting points for Hobbs problem
hobbs.f<- function(x){ # # Hobbs weeds problem -- function
    if (abs(12*x[3]) > 500) { # check computability
```

```
       fbad<-.Machine$double.xmax
       return(fbad)
    }
    res<-hobbs.res(x)
    f<-sum(res*res)
##    cat("fval =",f,"\n")
##    f
}
hobbs.res<-function(x){ # 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)
    t<-1:12
    if(abs(12*x[3])>50) {
       res<-rep(Inf,12)
    } else {
       res<-x[1]/(1+x[2]*exp(-x[3]*t)) - y
    }
}

hobbs.jac<-function(x){ # Jacobian of Hobbs weeds problem
   jj<-matrix(0.0, 12, 3)
   t<-1:12
    yy<-exp(-x[3]*t)
    zz<-1.0/(1+x[2]*yy)
     jj[t,1] <- zz
     jj[t,2] <- -x[1]*zz*zz*yy
     jj[t,3] <- x[1]*zz*zz*yy*x[2]*t
   return(jj)
}

hobbs.g<-function(x){ # gradient of Hobbs weeds problem
    # NOT EFFICIENT TO CALL AGAIN
    jj<-hobbs.jac(x)
    res<-hobbs.res(x)
    gg<-as.vector(2.*t(jj) %*% res)
    return(gg)
}
require(optimx)
set.seed(12345)
nrun<-100
sstart<-matrix(runif(3*nrun, 0, 5), nrow=nrun, ncol=3)
ustart<-sstart %*% diag(c(100, 10, 0.1))
nsuccR <- 0
nsuccO <- 0
vRvm <- rep(NA, nrun)
voptim <- vRvm
fRvm <- vRvm
gRvm <- vRvm
foptim <- vRvm
goptim <- vRvm
```

```
for (irun in 1:nrun) {
  us <- ustart[irun,]
#  print(us)
#  ans <- Rvmminu(us, hobbs.f, hobbs.g, control=list(trace=1))
#  ans <- optim(us, hobbs.f, hobbs.g, method="BFGS")
  ans <- Rvmminu(us, hobbs.f, hobbs.g, control=list(trace=0))
  ao <- optim(us, hobbs.f, hobbs.g, method="BFGS",
              control=list(maxit=3000))
# ensure does not max function out

# cat(irun,"  Rvmminu value =",ans$value,"  optim:BFGS value=",ao$value,"\n")
  if (ans$value < 2.5879) nsuccR <- nsuccR + 1
  if (ao$value < 2.5879) nsuccO <- nsuccO + 1
#  tmp <- readline()
  vRvm[irun] <- ans$value
  voptim[irun] <- ao$value
  fRvm[irun] <- ans$counts[1]
  gRvm[irun] <- ans$counts[2]
  foptim[irun] <- ao$counts[1]
  goptim[irun] <- ao$counts[2]

}
cat("Rvmminu: number of successes=",nsuccR,"  propn=",nsuccR/nrun,"\n")
```

```
## Rvmminu: number of successes= 99   propn= 0.99
```

```
cat("optim:BFGS no. of successes=",nsuccO,"  propn=",nsuccO/nrun,"\n")
```

```
## optim:BFGS no. of successes= 99   propn= 0.99
```

```
fgc <- data.frame(fRvm, foptim, gRvm, goptim)
summary(fgc)
```

```
##       fRvm            foptim            gRvm           goptim
##  Min.   : 37.00   Min.   :  58.0   Min.   :20.00   Min.   : 16.0
##  1st Qu.: 58.00   1st Qu.: 138.8   1st Qu.:31.75   1st Qu.: 53.0
##  Median : 77.00   Median : 182.0   Median :41.00   Median : 68.5
##  Mean   : 87.26   Mean   : 323.8   Mean   :40.78   Mean   :131.2
##  3rd Qu.: 94.25   3rd Qu.: 455.0   3rd Qu.:48.00   3rd Qu.:178.8
##  Max.   :856.00   Max.   :1427.0   Max.   :83.00   Max.   :610.0
```

From this summary, it appears that Rvmmin, on average, uses fewer gradient and function evaluations to achieve the desired result.

For comparison, we now re-run the example with default settings for maxit in optim:BFGS.

```
nsuccR <- 0
nsuccO <- 0
for (irun in 1:nrun) {
  us <- ustart[irun,]
#  print(us)
#  ans <- Rvmminu(us, hobbs.f, hobbs.g, control=list(trace=1))
#  ans <- optim(us, hobbs.f, hobbs.g, method="BFGS")
  ans <- Rvmminu(us, hobbs.f, hobbs.g, control=list(trace=0))
  ao <- optim(us, hobbs.f, hobbs.g, method="BFGS")
# ensure does not max function out
```

```
# cat(irun,"  Rvmminu value =",ans$value,"  optim:BFGS value=",ao$value,"\n")
  if (ans$value < 2.5879) nsuccR <- nsuccR + 1
  if (ao$value < 2.5879) nsuccO <- nsuccO + 1
#  tmp <- readline()
  vRvm[irun] <- ans$value
  voptim[irun] <- ao$value
  fRvm[irun] <- ans$counts[1]
  gRvm[irun] <- ans$counts[2]
  foptim[irun] <- ao$counts[1]
  goptim[irun] <- ao$counts[2]

}
cat("Rvmminu: number of successes=",nsuccR,"  propn=",nsuccR/nrun,"\n")
```

```
## Rvmminu: number of successes= 99   propn= 0.99
```

```
cat("optim:BFGS no. of successes=",nsuccO,"  propn=",nsuccO/nrun,"\n")
```

```
## optim:BFGS no. of successes= 64   propn= 0.64
```

```
fgc <- data.frame(fRvm, foptim, gRvm, goptim)
summary(fgc)
```

```
##       fRvm            foptim           gRvm           goptim
##  Min.   : 37.00   Min.   : 58.0   Min.   :20.00   Min.   : 16.00
##  1st Qu.: 58.00   1st Qu.:138.8   1st Qu.:31.75   1st Qu.: 53.00
##  Median : 77.00   Median :182.0   Median :41.00   Median : 68.50
##  Mean   : 87.26   Mean   :184.1   Mean   :40.78   Mean   : 71.73
##  3rd Qu.: 94.25   3rd Qu.:236.0   3rd Qu.:48.00   3rd Qu.:100.00
##  Max.   :856.00   Max.   :425.0   Max.   :83.00   Max.   :100.00
```

Bounds and masks

Let us make sure that Rvmminb is doing the right thing with bounds and masks. (This is actually a test in the package.)

Bounds

```
bt.f<-function(x){
 sum(x*x)
}

bt.g<-function(x){
  gg<-2.0*x
}

lower <- c(0, 1, 2, 3, 4)
upper <- c(2, 3, 4, 5, 6)
bdmsk <- rep(1,5)
xx <- rep(0,5) # out of bounds
ans <- Rvmmin(xx, bt.f, bt.g, lower=lower, upper=upper, bdmsk=bdmsk)
```

```
## Warning in Rvmmin(xx, bt.f, bt.g, lower = lower, upper = upper, bdmsk = bdmsk):
## Parameter out of bounds has been moved to nearest bound
```

```
ans
```

```
## $par
## [1] 0 1 2 3 4
## attr(,"status")
## [1] "L" "L" "L" "L" "L"
##
## $value
## [1] 30
##
## $counts
## function gradient
##        1        1
##
## $convergence
## [1] 0
##
## $message
## [1] "Rvmminb appears to have converged"
##
## $bdmsk
## [1]  1 -3 -3 -3 -3
```

Masks

Here we fix one or more paramters and minimize over the rest.

```
sq.f<-function(x){
   nn<-length(x)
   yy<-1:nn
   f<-sum((yy-x)^2)
   f
}
sq.g <- function(x){
   nn<-length(x)
   yy<-1:nn
   gg<- 2*(x - yy)
}

xx0 <- rep(pi,3)
bdmsk <- c(1, 0, 1) # Middle parameter fixed at pi
cat("Check final function value (pi-2)^2 = ", (pi-2)^2,"\n")
```

```
## Check final function value (pi-2)^2 =  1.303234
```

```
require(optimx)
ans <- Rvmmin(xx0, sq.f, sq.g, lower=-Inf, upper=Inf, bdmsk=bdmsk,
              control=list(trace=2))
```

```
## Bounds: nolower =  TRUE   noupper =  TRUE  bounds =  TRUE
## Initial parameters:[1] 3.141593 3.141593 3.141593
## admissible =  TRUE
## maskadded =  FALSE
## lower:[1] -1.797693e+308 -1.797693e+308 -1.797693e+308
## upper:[1] 1.797693e+308 1.797693e+308 1.797693e+308
## parchanged =  FALSE
```

```
## Bounds: nolower =  FALSE   noupper =  FALSE  bounds =  TRUE
## Rvmminb -- J C Nash 2009-2015 - an R implementation of Alg 21
## Problem of size n= 3   Dot arguments:
## list()
## Initial fn= 5.909701
## ig= 1   gnorm= 4.861975   Reset Inv. Hessian approx at ilast =  1
##   1   1   5.909701
## Gradproj = -18.42587
## reset steplength= 1
## *reset steplength= 0.2
## Parameter  1  is free
## Parameter  3  is free
## ig= 2   gnorm= 2.575522     3   2   2.961562
## Gradproj = -15.04576
## reset steplength= 1
## *reset steplength= 0.2
## Parameter  1  is free
## Parameter  3  is free
## ig= 3   gnorm= 0.23879     5   3   1.317489
## Gradproj = -0.02851034
## reset steplength= 1
## Parameter  1  is free
## Parameter  3  is free
## ig= 4   gnorm= 0   Small gradient norm
## Seem to be done Rvmminb
```

```
ans
```

```
## $par
## [1] 1.000000 3.141593 3.000000
##
## $value
## [1] 1.303234
##
## $counts
## function gradient
##        6        4
##
## $convergence
## [1] 2
##
## $message
## [1] "Rvmminb appears to have converged"
##
## $bdmsk
## [1] 1 0 1
```

```
ansnog <- Rvmmin(xx0, sq.f, lower=-Inf, upper=Inf, bdmsk=bdmsk,
               control=list(trace=2))
```

```
## Bounds: nolower =  TRUE   noupper =  TRUE  bounds =  TRUE
## WARNING: forward gradient approximation being used
## Initial parameters:[1] 3.141593 3.141593 3.141593
## admissible =  TRUE
## maskadded =  FALSE
## lower:[1] -1.797693e+308 -1.797693e+308 -1.797693e+308
```

```
## upper:[1] 1.797693e+308 1.797693e+308 1.797693e+308
## parchanged =  FALSE
## Bounds: nolower =  FALSE   noupper =  FALSE  bounds =  TRUE
## Rvmminb -- J C Nash 2009-2015 - an R implementation of Alg 21
## Problem of size n= 3   Dot arguments:
## list()
## WARNING: using gradient approximation ' grfwd '
## Initial fn= 5.909701
## ig= 1   gnorm= 10.41055   Reset Inv. Hessian approx at ilast =  1
##   1   1   5.909701
## Gradproj = -65.26225
## reset steplength= 1
## *reset steplength= 0.2
## Parameter  1  is free
## Parameter  3  is free
## ig= 2   gnorm= 5.538718     3   2   4.463114
## Gradproj = -222.3618
## reset steplength= 1
## *reset steplength= 0.2
## *reset steplength= 0.04
## *reset steplength= 0.008
## *reset steplength= 0.0016
## *reset steplength= 0.00032
## *reset steplength= 6.4e-05
## *reset steplength= 1.28e-05
## *reset steplength= 2.56e-06
## *reset steplength= 5.12e-07
## *reset steplength= 1.024e-07
## *reset steplength= 2.048e-08
## *reset steplength= 4.096e-09
## *reset steplength= 8.192e-10
## *reset steplength= 1.6384e-10
## *reset steplength= 3.2768e-11
## *reset steplength= 6.5536e-12
## *reset steplength= 1.31072e-12
## *reset steplength= 2.62144e-13
## *reset steplength= 5.24288e-14
## *reset steplength= 1.048576e-14
## *reset steplength= 2.097152e-15
## *reset steplength= 4.194304e-16
## *reset steplength= 8.388608e-17
## *reset steplength= 1.677722e-17
## Unchanged in step redn
## No acceptable point
## Reset to gradient search
## Reset Inv. Hessian approx at ilast =  2
##   27   2   4.463114
## Gradproj = -30.67739
## reset steplength= 1
## *reset steplength= 0.2
## *reset steplength= 0.04
## *reset steplength= 0.008
## *reset steplength= 0.0016
## *reset steplength= 0.00032
```

```
## *reset steplength= 6.4e-05
## *reset steplength= 1.28e-05
## *reset steplength= 2.56e-06
## *reset steplength= 5.12e-07
## *reset steplength= 1.024e-07
## *reset steplength= 2.048e-08
## *reset steplength= 4.096e-09
## *reset steplength= 8.192e-10
## *reset steplength= 1.6384e-10
## *reset steplength= 3.2768e-11
## *reset steplength= 6.5536e-12
## *reset steplength= 1.31072e-12
## *reset steplength= 2.62144e-13
## *reset steplength= 5.24288e-14
## *reset steplength= 1.048576e-14
## *reset steplength= 2.097152e-15
## *reset steplength= 4.194304e-16
## *reset steplength= 8.388608e-17
## *reset steplength= 1.677722e-17
## Unchanged in step redn
## No acceptable point
## Converged
## Seem to be done Rvmminb
```

```
ansnog
```

```
## $par
## [1] 2.284955 3.141593 1.771680
##
## $value
## [1] 4.463114
##
## $counts
## function gradient
##       51        2
##
## $convergence
## [1] 0
##
## $message
## [1] "Rvmminb appears to have converged"
##
## $bdmsk
## [1] 1 0 1
```

References

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

———. 2014. *Nonlinear Parameter Optimization Using R Tools.* Book. John Wiley & Sons: Chichester. http://www.wiley.com//legacy/wileychi/nash/.