SNewton: safeguarded Newton methods for function minimization

John C. Nash

2018-07-10

Safeguarded Newton algorithms

So-called **Newton** methods are among the most commonly mentioned in the solution of nonlinear equations or function minimization. However, as discussed in

https://en.wikipedia.org/wiki/Newton%27s_method#History,

the **Newton** or **Newton-Raphson** method as we know it today was not what either of its supposed originators knew.

This vignette discusses the development of simple safeguarded variants of the Newton method for function minimization in **R**. Note that there are some resources in **R** for solving nonlinear equations by Newton-like methods in the packages **nleqslv** and **pracma**.

The basic approach

If we have a function $f(x)$, with gradient $g(x)$ and second derivative (Hessian) $H(x)$ the first order condition for an extremum (min or max) is

$$g(x) = 0$$

To ensure a minimum, we want

$$H(x) > 0$$

The first order condition leads to a root-finding problem.

It turns out that x need not be a scalar. We can consider it to be a vector of parameters to be determined. This renders $g(x)$ a vector also, and $H(x)$ a matrix. The conditions of optimality then require a zero gradient and positive-definite Hessian.

The Newton approach to such equations is to provide a guess to the root x_try and to then solve the equation

$$H(x_t) * s = -g(x_t)$$

for the search vector s. We update x_t to $x_t + s$ and repeat until we have a very small gradient $g(x_t)$. If $H(x)$ is positive definite, we have a reasonable approximation to a (local) minimum.

Motivations

A particular interest in Newton-like methods its theoretical quadratic convergence. See https://en.wikipedia.org/wiki/Newton%27s_method. That is, the method will converge in one step for a quadratic function $f(x)$, and for "reasonable" functions will converge very rapidly. There are, however, a number of conditions, and practical programs need to include safequards against mis-steps in the iterations.

The principal issues concern the possiblity that $H(x)$ may not be positive definite, at least in some parts of the domain, and that the curvature may be such that a unit step $x_t + s$ does not reduce the function f. We therefore get a number of possible variants of the method when different possible safeguards are applied.

Algorithm possibilities

There are many choices we can make in building a practical code to implement the ideas above. In tandem with the two main issues expressed above, we will consider

- the modification of the solution of the main equation

  $$H(x_t) * s = -g(x_t)$$

  so that a reasonable search vector s is always generated by avoiding Hessian matrices that are not positive definite.
- the selection of a new set of parameters $x_new = x_t + step * s$ so that the function value $f(x_new)$ is less than $f(x_t)$.

The second choice above could be made slightly more stringent so that the Armijo condition of sufficient-decrease is met. Adding a curvature requirement gives the Wolfe condisions. See https://en.wikipedia.org/wiki/Wolfe_conditions. The Armijo requirement is generally written

$$f(x_t + step * s) < f(x_t) + c * step * g(x_t)^T * s$$

where c is some number less than 1. Typically $c = 1e-4 = 0.0001$. Note that the product of gradient times search vector is negative for any reasonable situation, since we are trying to go "downhill".

As a result of the ideas in this section, the code `snewton()` uses a solution of the Newton equations with the Hessian provided (if this is possible, else we stop), along with a backtracking line search. The code `snewtonm` uses a Marquardt stabilization of the Hessian to create

$$Haug = H + 1_n * lambda$$

That is, we add *lambda* times the unit matrix to H. Then we try the set of parameters found by adding the solution of the Newton equations with $Haug$ in place of H to the current "best" set of parameters. If this new set of parameters has a higher function value than the "best" so far, we increase *lambda* and try again. Note that we do not need to re-evaluate the gradient or Hessian to do this. Moreover, for some value of *lambda*, the step is clearly down the gradient (i.e., steepest descents) or we have converged and no progress is possible. This leads to a very compact and elegant code, which we name `snewtonm()` for Safeguarded Newton-Marquardt method. It is reliable, but may be less efficient than using the un-modified Hessian.

A choice to compute the search vector

The primary concern in solving for s is that the Hessian may not be positive definite. This means that we cannot apply fast and stable methods like the Cholesky decomposition to the matrix. At the time of writing, we use the following approach:

- We attempt to solve

$$H(x_t) * s = -g(x_t)$$

with **R** directly, and rely on internal checks to catch any cases where the solution fails. We then use `try()` to stop the program in this case.

Choosing the step size

The traditional Newton approach is that the stepsize is taken to be 1. In practice, this can sometimes mean that the function value is not reduced. As an alternative, we can use a simple backtrack search. We start with $step = 1$ (actually the program allows for the element `defstep` of the `control` list to be set to a value other than 1). If the Armijo condition is not met, we replace *step* with $ r * step $ where r is less than 1. Here we suggest `control$stepdec = 0.2`. We repeat until x_t satisfies the Armijo condition or x_t is essentially unchanged by the step.

Here "essentially unchanged" is determined by a test using an offset value, that is, the test

$$(x_t + offset) == (x_t + step * d + offset)$$

where d is the search direction. `control$offset = 100` is used. We could also, and almost equivalently, use the **R** `identical` function.

This approach has been coded into the `snewton()` function.

Examples

These examples are coded as a test to the interim package **snewton**, but as at 2018-7-10 are part of the `optimx` package. We call these below mostly via the `optimr()` function to allow compact output to be used, but please note that some count information on the number of hessian evaluations and "iterations" (which generally is an algorithm-specific measure) is not then returned.

A simple example

The following example is trivial, in that the Hessian is a constant matrix, and we achieve convergence immediately.

```
x0<-c(1,2,3,4)
fnt <- function(x, fscale=10){
  yy <- length(x):1
  val <- sum((yy*x)^2)*fscale
}
grt <- function(x, fscale=10){
  nn <- length(x)
  yy <- nn:1
  #    gg <- rep(NA,nn)
  gg <- 2*(yy^2)*x*fscale
  gg
}

hesst <- function(x, fscale=10){
```

```
  nn <- length(x)
  yy <- nn:1
  hh <- diag(2*yy^2*fscale)
  hh
}

require(optimx)
```

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

```
t1 <- snewton(x0, fnt, grt, hesst, control=list(trace=0), fscale=3.0)
print(t1)
```

```
## $par
## [1] 0 0 0 0
##
## $value
## [1] 0
##
## $grad
## [1] 0 0 0 0
##
## $Hess
##      [,1] [,2] [,3] [,4]
## [1,]   96    0    0    0
## [2,]    0   54    0    0
## [3,]    0    0   24    0
## [4,]    0    0    0    6
##
## $counts
## $counts$niter
## [1] 2
##
## $counts$nfn
## [1] 2
##
## $counts$ngr
## [1] 2
##
## $counts$nhess
## [1] 1
##
##
## $convcode
## [1] 0
##
## $message
## [1] "Small gradient norm"
```

```
# we can also use nlm and nlminb
fght <- function(x, fscale=10){
  ## combine f, g and h into single function for nlm
     ff <- fnt(x, fscale)
     gg <- grt(x, fscale)
     hh <- hesst(x, fscale)
```

```
    attr(ff, "gradient") <- gg
    attr(ff, "hessian") <- hh
    ff
}

t1nlm <- nlm(fght, x0, fscale=3.0, hessian=TRUE, print.level=0)
print(t1nlm)
```

```
## $minimum
## [1] 1.232595e-29
##
## $estimate
## [1] 0.000000e+00 2.220446e-16 4.440892e-16 0.000000e+00
##
## $gradient
## [1] 0.000000e+00 3.996803e-14 3.552714e-14 0.000000e+00
##
## $hessian
##              [,1] [,2] [,3]         [,4]
## [1,] 3.200000e+02    0    0 4.235165e-14
## [2,] 0.000000e+00  180    0 0.000000e+00
## [3,] 0.000000e+00    0   80 0.000000e+00
## [4,] 4.235165e-14    0    0 2.000000e+01
##
## $code
## [1] 1
##
## $iterations
## [1] 1
```

```
## BUT ... it looks like nlminb is NOT using a true Newton-type method
t1nlminb <- nlminb(x0, fnt, gradient=grt, hessian=hesst, fscale=3.0,
                   control=list(trace=0))
print(t1nlminb)
```

```
## $par
## [1] -4.043175e-174   0.000000e+00 -3.234540e-173 -1.293816e-172
##
## $objective
## [1] 0
##
## $convergence
## [1] 0
##
## $iterations
## [1] 13
##
## $evaluations
## function gradient
##       15       13
##
## $message
## [1] "relative convergence (4)"
```

```
# and call them from optimx (i.e., test this gives same results)
t1so <-  optimr(x0, fnt, grt, hess=hesst, method="snewton", fscale=3.0,
                control=list(trace=0))
proptimr(t1so)
```

```
## Result  t1so   proposes optimum function value = 0  at parameters
## [1] 0 0 0 0
## After  2  fn evals, and  2  gr evals
## Termination code is  0 : Small gradient norm
## ----------------------------------------
```

```
t1nlmo <- optimr(x0, fnt, grt, hess=hesst, method="nlm", fscale=3.0,
                control=list(trace=0))
proptimr(t1nlmo)
```

```
## Result  t1nlmo   proposes optimum function value = 1.232595e-29  at parameters
## [1] 0.000000e+00 2.220446e-16 4.440892e-16 0.000000e+00
## After  NA  fn evals, and  1  gr evals
## Termination code is  0 : nlm: Convergence indicator (code) =  1
## ----------------------------------------
```

```
tst <- try(t1nlminbo <- optimr(x0, fnt, grt, hess=hesst, method="nlminb",
                               fscale=3.0, control=list(trace=0)))
if (class(tst) == "try-error"){
    cat("try-error on attempt to run nlminb in optimr()\n")
} else { proptimr(t1nlminbo) }
```

```
## Result  t1nlminbo   proposes optimum function value = 0  at parameters
## [1] -4.043175e-174   0.000000e+00 -3.234540e-173 -1.293816e-172
## After  15  fn evals, and  13  gr evals
## Termination code is  0 : relative convergence (4)
## ----------------------------------------
```

From the number of function and gradient evaluations, it appears `nlminb()` is not using the Hessian information. Note that the `snewton()` and `snewtonm()` functions return count information for iterations and hessian evaluations, but these are not part of the standard `optim()` (and thus `optimr()`) result objects.

The Rosenbrock function

```
require(optimx)
#Rosenbrock banana valley function
f <- function(x){
return(100*(x[2] - x[1]*x[1])^2 + (1-x[1])^2)
}
#gradient
gr <- function(x){
return(c(-400*x[1]*(x[2] - x[1]*x[1]) - 2*(1-x[1]), 200*(x[2] - x[1]*x[1])))
}
#Hessian
h <- function(x) {
a11 <- 2 - 400*x[2] + 1200*x[1]*x[1]; a21 <- -400*x[1]
return(matrix(c(a11, a21, a21, 200), 2, 2))
}
x0 <- c(-1.2, 1)
```

```
# sink("mbrn1-170408.txt", split=TRUE)
t1 <- snewton(x0, fn=f, gr=gr, hess=h, control=list(trace=0))
print(t1)
```

```
## $par
## [1] 1 1
##
## $value
## [1] 0
##
## $grad
## [1] 0 0
##
## $Hess
##      [,1] [,2]
## [1,]  802 -400
## [2,] -400  200
##
## $counts
## $counts$niter
## [1] 25
##
## $counts$nfn
## [1] 33
##
## $counts$ngr
## [1] 25
##
## $counts$nhess
## [1] 24
##
##
## $convcode
## [1] 0
##
## $message
## [1] "Small gradient norm"
```

```
# we can also use nlm and nlminb
fght <- function(x){
  ## combine f, g and h into single function for nlm
    ff <- f(x)
    gg <- gr(x)
    hh <- h(x)
    attr(ff, "gradient") <- gg
    attr(ff, "hessian") <- hh
    ff
}

# COULD TRY: t1nlm <- nlm(fght, x0, hessian=TRUE, print.level=2, iterlim=10000)
t1nlmo <- optimr(x0, f, gr, hess=h, method="nlm", control=list(trace=0))
proptimr(t1nlmo)
```

```
## Result  t1nlmo   proposes optimum function value = 1.127026e-17  at parameters
```

```
## [1] 1 1
## After  NA  fn evals, and  24  gr evals
## Termination code is  0 : nlm: Convergence indicator (code) =  1
## ----------------------------------------
```

```
t1so <- optimr(x0, f, gr, hess=h, method="snewton", control=list(trace=0))
proptimr(t1so)
```

```
## Result  t1so   proposes optimum function value = 0  at parameters
## [1] 1 1
## After  33  fn evals, and  25  gr evals
## Termination code is  0 : Small gradient norm
## ----------------------------------------
```

```
t1smo <-  optimr(x0, f, gr, hess=h, method="snewtonm", control=list(trace=0))
```

```
## Small gradient
```

```
proptimr(t1smo)
```

```
## Result  t1smo   proposes optimum function value = 0  at parameters
## [1] 1 1
## After  46  fn evals, and  28  gr evals
## Termination code is  0 : snewtonm: Normal exit
## ----------------------------------------
```

```
## Again, nlminb probably not using hessian
tst <- try(t1nlminbo <- optimr(x0, f, gr, hess=h, method="nlminb",
                               control=list(trace=0)))
if (class(tst) == "try-error"){
    cat("try-error on attempt to run nlminb in optimr()\n")
} else { proptimr(t1nlminbo) }
```

```
## Result  t1nlminbo   proposes optimum function value = 0  at parameters
## [1] 1 1
## After  33  fn evals, and  26  gr evals
## Termination code is  0 : X-convergence (3)
## ----------------------------------------
```

The Wood function

For `nlm()` the "standard" start takes more than 100 iterations and returns a non-optimal solution.

```
#Example: Wood function
#
wood.f <- function(x){
  res <- 100*(x[1]^2-x[2])^2+(1-x[1])^2+90*(x[3]^2-x[4])^2+(1-x[3])^2+
    10.1*((1-x[2])^2+(1-x[4])^2)+19.8*(1-x[2])*(1-x[4])
  return(res)
}
#gradient:
wood.g <- function(x){
  g1 <- 400*x[1]^3-400*x[1]*x[2]+2*x[1]-2
  g2 <- -200*x[1]^2+220.2*x[2]+19.8*x[4]-40
  g3 <- 360*x[3]^3-360*x[3]*x[4]+2*x[3]-2
  g4 <- -180*x[3]^2+200.2*x[4]+19.8*x[2]-40
  return(c(g1,g2,g3,g4))
```

```
}
#hessian:
wood.h <- function(x){
  h11 <- 1200*x[1]^2-400*x[2]+2;    h12 <- -400*x[1]; h13 <- h14 <- 0
  h22 <- 220.2; h23 <- 0;    h24 <- 19.8
  h33 <- 1080*x[3]^2-360*x[4]+2;    h34 <- -360*x[3]
  h44 <- 200.2
  H <- matrix(c(h11,h12,h13,h14,h12,h22,h23,h24,
                h13,h23,h33,h34,h14,h24,h34,h44),ncol=4)
  return(H)
}

wood.fgh <- function(x){
      fval <- wood.f(x)
      gval <- wood.g(x)
      hval <- wood.h(x)
      attr(fval,"gradient") <- gval
      attr(fval,"hessian")<- hval
      fval
}

####################################################
x0 <- c(-3,-1,-3,-1) # Wood standard start

require(optimx)
# In 100 iterations, not converged
t1nlm <- nlm(wood.fgh, x0, print.level=0)
print(t1nlm)
```

```
## $minimum
## [1] 7.874467
##
## $estimate
## [1] -1.0316067  1.0740774 -0.9012710  0.8239815
##
## $gradient
## [1]  0.007517820 -0.015797320 -0.008937001  0.015745111
##
## $code
## [1] 4
##
## $iterations
## [1] 100
```

```
# But both newton approaches do work
wd <- snewton(x0, fn=wood.f, gr=wood.g, hess=wood.h, control=list(trace=0))
print(wd)
```

```
## $par
## [1] 1 1 1 1
##
## $value
## [1] 1.142616e-29
##
```

```
## $grad
## [1] -2.442491e-15 -7.105427e-15  3.108624e-15  0.000000e+00
##
## $Hess
##      [,1]   [,2] [,3]   [,4]
## [1,]  802 -400.0    0    0.0
## [2,] -400  220.2    0   19.8
## [3,]    0    0.0  722 -360.0
## [4,]    0   19.8 -360  200.2
##
## $counts
## $counts$niter
## [1] 70
##
## $counts$nfn
## [1] 119
##
## $counts$ngr
## [1] 70
##
## $counts$nhess
## [1] 70
##
##
## $convcode
## [1] 92
##
## $message
## [1] "No progress before linesearch!"
```

```
wdm <- snewtonm(x0, fn=wood.f, gr=wood.g, hess=wood.h, control=list(trace=0))
print(wdm)
```

```
## $par
## [1] 1 1 1 1
##
## $value
## [1] 1.399599e-28
##
## $grad
## [1] -1.243450e-14  3.552714e-14  1.243450e-14  0.000000e+00
##
## $Hess
##      [,1]   [,2] [,3]   [,4]
## [1,]  802 -400.0    0    0.0
## [2,] -400  220.2    0   19.8
## [3,]    0    0.0  722 -360.0
## [4,]    0   19.8 -360  200.2
##
## $counts
## $counts$niter
## [1] 89
##
## $counts$nfn
## [1] 88
```

```
##
## $counts$ngr
## [1] 50
##
## $counts$nhess
## [1] 50
##
##
## $convcode
## [1] 0
##
## $message
## [1] "snewtonm: Normal exit"
```

```
## AND again nlminb not likely using hessian information
## t1nlminb <- nlminb(x0, wood.f, gradient=wood.g, hess=wood.h, control=list(trace=0))
## print(t1nlminb)
# and call them from optimx (i.e., test this gives same results)

# But optimr uses a larger iteration limit, and gets to solution
t1nlmo <- optimr(x0, wood.f, wood.g, hess=wood.h, method="nlm", control=list(trace=0))
proptimr(t1nlmo)
```

```
## Result  t1nlmo   proposes optimum function value = 1.004943e-16  at parameters
## [1] 1 1 1 1
## After  NA  fn evals, and  335  gr evals
## Termination code is  0 : nlm: Convergence indicator (code) =  1
## ----------------------------------------
```

```
tst<-try(t1nlminbo <- optimr(x0, wood.f, wood.g, hess=wood.h, method="nlminb", control=list(trace=0)))
if (class(tst) == "try-error"){
    cat("try-error on attempt to run nlminb in optimr()\n")
} else { proptimr(t1nlminbo) }
```

```
## Result  t1nlminbo   proposes optimum function value = 4.79233e-29  at parameters
## [1] 1 1 1 1
## After  56  fn evals, and  45  gr evals
## Termination code is  0 : X-convergence (3)
## ----------------------------------------
```

A generalized Rosenbrock function

There are several generalizations of the Rosenbrock function (??ref)

```
# genrosa function code -- attempts to match the rosenbrock at gs=100 and x=c(-1.2,1)
genrosa.f<- function(x, gs=NULL){ # objective function
## One generalization of the Rosenbrock banana valley function (n parameters)
    n <- length(x)
        if(is.null(gs)) { gs=100.0 }
        # Note do not at 1.0 so min at 0
    fval<-sum (gs*(x[1:(n-1)]^2 - x[2:n])^2 + (x[1:(n-1)] - 1)^2)
}

genrosa.g <- function(x, gs=NULL){
# vectorized gradient for genrose.f
```

```
# Ravi Varadhan 2009-04-03
    n <- length(x)
        if(is.null(gs)) { gs=100.0 }
    gg <- as.vector(rep(0, n))
    tn <- 2:n
    tn1 <- tn - 1
    z1 <- x[tn] - x[tn1]^2
    z2 <- 1 - x[tn1]
        # f = gs*z1*z1 + z2*z2
    gg[tn] <- 2 * (gs * z1)
    gg[tn1] <- gg[tn1] - 4 * gs * x[tn1] * z1 - 2 *z2
    return(gg)
}

genrosa.h <- function(x, gs=NULL) { ## compute Hessian
   if(is.null(gs)) { gs=100.0 }
    n <- length(x)
    hh<-matrix(rep(0, n*n),n,n)
    for (i in 2:n) {
        z1<-x[i]-x[i-1]*x[i-1]
#       z2<-1.0 - x[i-1]
                hh[i,i]<-hh[i,i]+2.0*(gs+1.0)
                hh[i-1,i-1]<-hh[i-1,i-1]-4.0*gs*z1-4.0*gs*x[i-1]*(-2.0*x[i-1])
                hh[i,i-1]<-hh[i,i-1]-4.0*gs*x[i-1]
                hh[i-1,i]<-hh[i-1,i]-4.0*gs*x[i-1]
    }
        return(hh)
}

require(optimx)
cat("Generalized Rosenbrock tests\n")
```

```
## Generalized Rosenbrock tests
```

```
cat("original n and x0")
```

```
## original n and x0
```

```
x0 <- c(-1.2, 1)
solorig <- snewton(x0, genrosa.f, genrosa.g, genrosa.h)
print(solorig)
```

```
## $par
## [1] 1 1
##
## $value
## [1] 2.972526e-28
##
## $grad
## [1]  5.462297e-14 -4.440892e-14
##
## $Hess
##      [,1] [,2]
## [1,]  800 -400
## [2,] -400  202
```

```
##
## $counts
## $counts$niter
## [1] 128
##
## $counts$nfn
## [1] 143
##
## $counts$ngr
## [1] 128
##
## $counts$nhess
## [1] 128
##
##
## $convcode
## [1] 92
##
## $message
## [1] "No progress before linesearch!"
```

```
print(eigen(solorig$Hess)$values)
```

```
## [1] 1000.400641    1.599359
```

```
solorigm <- snewtonm(x0, genrosa.f, genrosa.g, genrosa.h)
print(solorigm)
```

```
## $par
## [1] 1 1
##
## $value
## [1] 1.232595e-30
##
## $grad
## [1] -2.220446e-15  0.000000e+00
##
## $Hess
##      [,1] [,2]
## [1,]  800 -400
## [2,] -400  202
##
## $counts
## $counts$niter
## [1] 152
##
## $counts$nfn
## [1] 151
##
## $counts$ngr
## [1] 132
##
## $counts$nhess
## [1] 132
##
```

```
##
## $convcode
## [1] 0
##
## $message
## [1] "snewtonm: Normal exit"
```

```
print(eigen(solorigm$Hess)$values)
```

```
## [1] 1000.400641    1.599359
```

```
cat("Start with 50 values of pi and scale factor 10\n")
```

```
## Start with 50 values of pi and scale factor 10
```

```
x0 <- rep(pi, 50)
sol50pi <- optimr(x0, genrosa.f, genrosa.g, genrosa.h, method="snewton", gs=10)
proptimr(sol50pi)
```

```
## Result  sol50pi   proposes optimum function value = 6.108742e-29  at parameters
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## After  145  fn evals, and  145  gr evals
## Termination code is  92 : No progress before linesearch!
## ----------------------------------------
```

```
hhi <- genrosa.h(sol50pi$par, gs=10)
print(eigen(hhi)$values)
```

```
##  [1] 181.84200 181.36863 180.58176 179.48449 178.08116 176.37730 174.37964
##  [8] 172.09607 169.53560 166.70834 163.62545 160.29911 156.74243 152.96948
## [15] 148.99513 144.83509 140.50578 136.02429 131.40832 126.67610 121.84632
## [22] 116.93804 111.97066 106.96381 101.93725  96.91085  91.90447  86.93791
## [29]  82.03080  77.20253  72.47223  67.85859  63.37989  59.05387  54.89766
## [36]  50.92776  47.15992  43.60907  40.28933  37.21385  34.39481  31.84332
## [43]  29.56937  27.58175  25.88797  24.49427  23.40556  22.62547  22.15648
## [50]   2.00000
```

```
sol50pim <- optimr(x0, genrosa.f, genrosa.g, genrosa.h, method="snewtonm", gs=10)
proptimr(sol50pim)
```

```
## Result  sol50pim   proposes optimum function value = 1.528418e-30  at parameters
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## After  157  fn evals, and  152  gr evals
## Termination code is  0 : snewtonm: Normal exit
## ----------------------------------------
```

```
hhm <- genrosa.h(sol50pim$par, gs=10)
print(eigen(hhm)$values)
```

```
##  [1] 181.84200 181.36863 180.58176 179.48449 178.08116 176.37730 174.37964
##  [8] 172.09607 169.53560 166.70834 163.62545 160.29911 156.74243 152.96948
## [15] 148.99513 144.83509 140.50578 136.02429 131.40832 126.67610 121.84632
## [22] 116.93804 111.97066 106.96381 101.93725  96.91085  91.90447  86.93791
## [29]  82.03080  77.20253  72.47223  67.85859  63.37989  59.05387  54.89766
## [36]  50.92776  47.15992  43.60907  40.28933  37.21385  34.39481  31.84332
## [43]  29.56937  27.58175  25.88797  24.49427  23.40556  22.62547  22.15648
```

```
## [50]   2.00000
```

The Hobbs weed infestation problem

This problem is described in @cnm79. It has various nasty properties. Note that one starting point causes failure of the `snewton()` optimizer.

```
## Optimization test function HOBBS
## ?? refs (put in .doc??)
## Nash and Walker-Smith (1987, 1989) ...
require(optimx)

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)
}


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)
}
```

```
hobbs.rsd<-function(x) { # Jacobian second derivative
    rsd<-array(0.0, c(12,3,3))
    t<-1:12
    yy<-exp(-x[3]*t)
    zz<-1.0/(1+x[2]*yy)
    rsd[t,1,1]<- 0.0
    rsd[t,2,1]<- -yy*zz*zz
    rsd[t,1,2]<- -yy*zz*zz
    rsd[t,2,2]<- 2.0*x[1]*yy*yy*zz*zz*zz
    rsd[t,3,1]<- t*x[2]*yy*zz*zz
    rsd[t,1,3]<- t*x[2]*yy*zz*zz
    rsd[t,3,2]<- t*x[1]*yy*zz*zz*(1-2*x[2]*yy*zz)
    rsd[t,2,3]<- t*x[1]*yy*zz*zz*(1-2*x[2]*yy*zz)
##    rsd[t,3,3]<- 2*t*t*x[1]*x[2]*x[2]*yy*yy*zz*zz*zz
    rsd[t,3,3]<- -t*t*x[1]*x[2]*yy*zz*zz*(1-2*yy*zz*x[2])
    return(rsd)
}


hobbs.h <- function(x) { ## compute Hessian
#   cat("Hessian not yet available\n")
#   return(NULL)
    H<-matrix(0,3,3)
    res<-hobbs.res(x)
    jj<-hobbs.jac(x)
    rsd<-hobbs.rsd(x)
##    H<-2.0*(t(res) %*% rsd + t(jj) %*% jj)
    for (j in 1:3) {
       for (k in 1:3) {
          for (i in 1:12) {
             H[j,k]<-H[j,k]+res[i]*rsd[i,j,k]
          }
       }
    }
    H<-2*(H + t(jj) %*% jj)
    return(H)
}

require(optimx)
x0 <- c(200, 50, .3)
cat("Start for Hobbs:")
```

```
## Start for Hobbs:
```

```
print(x0)
```

```
## [1] 200.0  50.0   0.3
```

```
solx0 <- snewton(x0, hobbs.f, hobbs.g, hobbs.h)
## Note that we exceed count limit, but have answer
print(solx0)
```

```
## $par
## [1] 196.1862618  49.0916395   0.3135697
##
```

```
## $value
## [1] 2.587277
##
## $grad
## [1]  5.101475e-14 -1.320055e-13  6.343726e-11
##
## $Hess
##             [,1]         [,2]        [,3]
## [1,]    1.265461    -3.256125    1602.105
## [2,]   -3.256125     8.627095   -4095.206
## [3,] 1602.105263 -4095.206388 2043434.033
##
## $counts
## $counts$niter
## [1] 501
##
## $counts$nfn
## [1] 501
##
## $counts$ngr
## [1] 501
##
## $counts$nhess
## [1] 500
##
##
## $convcode
## [1] 1
##
## $message
## [1] "Snewton - no msg"
```

```
print(eigen(solx0$Hess)$values)
```

```
## [1] 2.043443e+06 4.249248e-01 4.413953e-03
```

```
## Note that we exceed count limit, but have answer

## Setting relative check offset larger gets quicker convergence
solx0a <- snewton(x0, hobbs.f, hobbs.g, hobbs.h, control=list(offset=1000.))
print(solx0a)
```

```
## $par
## [1] 196.1862618  49.0916395   0.3135697
##
## $value
## [1] 2.587277
##
## $grad
## [1] -1.071365e-14  2.819966e-14 -1.364242e-11
##
## $Hess
##             [,1]         [,2]        [,3]
## [1,]    1.265461    -3.256125    1602.105
## [2,]   -3.256125     8.627095   -4095.206
```

```
## [3,] 1602.105263 -4095.206388 2043434.033
##
## $counts
## $counts$niter
## [1] 13
##
## $counts$nfn
## [1] 13
##
## $counts$ngr
## [1] 13
##
## $counts$nhess
## [1] 13
##
##
## $convcode
## [1] 92
##
## $message
## [1] "No progress before linesearch!"
```

```
x1s <- c(100, 10, .1)
cat("Start for Hobbs:")
```

```
## Start for Hobbs:
```

```
print(x1s)
```

```
## [1] 100.0  10.0   0.1
```

```
solx1s <- snewton(x1s, hobbs.f, hobbs.g, hobbs.h, control=list(trace=0))
print(solx1s)
```

```
## $par
## [1] 196.1862618  49.0916395   0.3135697
##
## $value
## [1] 2.587277
##
## $grad
## [1]  4.268808e-14 -1.088019e-13  5.439915e-11
##
## $Hess
##             [,1]         [,2]        [,3]
## [1,]    1.265461    -3.256125    1602.105
## [2,]   -3.256125     8.627095   -4095.206
## [3,] 1602.105263 -4095.206388 2043434.033
##
## $counts
## $counts$niter
## [1] 22
##
## $counts$nfn
## [1] 31
##
```

```
## $counts$ngr
## [1] 22
##
## $counts$nhess
## [1] 22
##
##
## $convcode
## [1] 92
##
## $message
## [1] "No progress before linesearch!"
```

```
print(eigen(solx1s$Hess)$values)
```

```
## [1] 2.043443e+06 4.249248e-01 4.413953e-03
```

```
solx1m <- snewton(x1s, hobbs.f, hobbs.g, hobbs.h, control=list(trace=0))
print(solx1m)
```

```
## $par
## [1] 196.1862618  49.0916395   0.3135697
##
## $value
## [1] 2.587277
##
## $grad
## [1]  4.268808e-14 -1.088019e-13  5.439915e-11
##
## $Hess
##             [,1]         [,2]        [,3]
## [1,]    1.265461    -3.256125    1602.105
## [2,]   -3.256125     8.627095   -4095.206
## [3,] 1602.105263 -4095.206388 2043434.033
##
## $counts
## $counts$niter
## [1] 22
##
## $counts$nfn
## [1] 31
##
## $counts$ngr
## [1] 22
##
## $counts$nhess
## [1] 22
##
##
## $convcode
## [1] 92
##
## $message
## [1] "No progress before linesearch!"
```

```
print(eigen(solx1m$Hess)$values)
```

```
## [1] 2.043443e+06 4.249248e-01 4.413953e-03
```

```
cat("Following test fails in snewton with ERROR -- Why?\n")
```

```
## Following test fails in snewton with ERROR -- Why?
```

```
x3 <- c(1, 1, 1)
cat("Start for Hobbs:")
```

```
## Start for Hobbs:
```

```
print(x3)
```

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

```
ftest <- try(solx3 <- snewton(x3, hobbs.f, hobbs.g, hobbs.h, control=list(trace=0)))
if (class(ftest) != "try-error") {
   print(solx3)
   print(eigen(solx3$Hess)$values)
}
```

```
## $par
## [1] 1 1 1
##
## $value
## [1] 23520.58
##
## $grad
## [1] -824.042084    4.764888  -11.025384
##
## $Hess
##           [,1]      [,2]      [,3]
## [1,] 22.321687  4.150703 -9.853740
## [2,]  4.150703 -1.275310 -9.255147
## [3,] -9.853740 -9.255147 33.506883
##
## $counts
## $counts$niter
## [1] 1
##
## $counts$nfn
## [1] 2
##
## $counts$ngr
## [1] 1
##
## $counts$nhess
## [1] 1
##
##
## $convcode
## [1] 9999
##
## $message
## [1] "snewton: New function value infinite"
```

```
##
## [1] 41.618914 16.635191 -3.700846
```

```
cat("But Marquardt variant succeeds\n")
```

```
## But Marquardt variant succeeds
```

```
solx3m <- snewtonm(x3, hobbs.f, hobbs.g, hobbs.h, control=list(trace=0))
print(solx3m)
```

```
## $par
## [1] 196.1862618  49.0916395   0.3135697
##
## $value
## [1] 2.587277
##
## $grad
## [1] -2.675637e-14  6.739054e-14 -3.473133e-11
##
## $Hess
##             [,1]         [,2]        [,3]
## [1,]    1.265461    -3.256125    1602.105
## [2,]   -3.256125     8.627095   -4095.206
## [3,] 1602.105263 -4095.206388 2043434.033
##
## $counts
## $counts$niter
## [1] 49
##
## $counts$nfn
## [1] 48
##
## $counts$ngr
## [1] 28
##
## $counts$nhess
## [1] 28
##
##
## $convcode
## [1] 0
##
## $message
## [1] "snewtonm: Normal exit"
```

```
print(eigen(solx3m$Hess)$values)
```

```
## [1] 2.043443e+06 4.249248e-01 4.413953e-03
```

```
# we can also use nlm and nlminb and call them from optimx
```