SNewton: safeguarded Newton methods for function minimization

John C. Nash

2023-06-21

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**. These are intended as learning tools, though the Marquardt stabilized version appears to be quite efficient. Note that there are some resources in **R** for solving nonlinear equations by Newton-like methods in the packages **nleqslv** and **pracma**. Also the base-R functions **nlminb()** and **nlm()** can make use of Hessians if provided, as can tools in the **trust** package.

The basic approach

If we have a function of one variable $f(x)$, 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_t and to then solve the **Newton equations**

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

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 are their 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 **safeguards** against mis-steps in the iterations. Such mis-steps occur because finite-precision floating-point arithmetic incurs errors, particularly when there are numbers of vastly different scale involved, or else implicit assumptions such as continuity of functions or their derivatives are not true.

One principal issue concerns the possibility 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 approach when different possible safeguards are applied, and different methods for (possibly approximate) solution of the Newton equations are used.

Some 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".

A safeguarded Newton method

As a result of the ideas above, 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**, where we reduce the step size until the Armijo condition is met or terminate with the suggestion that the current x_t is our solution. Note that Hartley (1961) suggested evaluating the function at $x_t + s$ and $x_t + 0.5 * s$ to provide three values that can be used for a parabolic inverse interpolation. However, a back-tracking search with acceptable point criterion is generally simpler yet still effective.

Newton-Marquardt method

A slightly different approach in the code `snewtonm` uses a Marquardt stabilization of the Hessian to create

$$H_{aug} = H + 1_n * lambda$$

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 H_{aug} 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 large value of *lambda*, the step is clearly almost 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.

Note that it is also possible to combine the Marquardt stabilization with a line search. Thus there is a multitude of possible methods in this general family, which can lead to potential disagreements about which are "best" unless there is great care taken to ensure the methods under discussion are well-defined.

Computing the search vector

If, when solving solving for s in the Newton equations, the Hessian is not positive definite, we cannot apply fast and stable methods like the Cholesky decomposition. The Newton-Marquardt for sufficiently large λ avoids this difficulty.

However, the solution often DOES work, and we can simply try to solve, indeed, wrapping the solution statement in the R `try()` function, and stop in the event of failure.

Choosing the step size in the safeguarded Newton method

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 generally start with $step = 1$ but it is trivial to allow for a smaller or bigger value. Indeed, the `control` list element `defstep` in the program `snewton` allows the initial step 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. Experience has shown it to be a rather poor method.

Bounds and masks constraints

In late 2021, the addition of bounds and masks constraints to `snewtonm` was begun. This uses the approach described in the vignette **Explaining Gradient Minimizers in R**. The function `snewtonmb()` was developed, and it was discovered that simply bypassing code for bounds allowed it to run about as quickly as the original (unconstrained) `snewtonm()` routine, which it now replaces, since there seems no merit in maintaining two routines.

The same ideas could be applied to `snewton()`, but my opinion is that the Marquardt stabilization gives `snewtonm()` an advantage in reliably finding solutions because the search direction is modified to guarantee a descent direction in the latter code.

Examples

These examples were 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, since this lets us include the "fname" attribute in the output of function `proptimr()`. Note that some count information on the number of hessian evaluations and "iterations" (which generally is an algorithm-specific measure) is not always provided.

A simple example

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

```
# SimpHess.R
x0<-c(1,2,3,4)
lo <- x0-0.5
up <- x0+1.0
fnt <- function(x, fscale=10){
  yy <- length(x):1
  val <- sum((yy*x)^2)*fscale
  val
}
attr(fnt,"fname")<-"SimpHess"
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 <- optimr(x0, fnt, grt, hesst, method="snewton", control=list(trace=0, usexxxmeth=TRUE), fscale=3.0)
```

```
## Warning in optimr(x0, fnt, grt, hesst, method = "snewton", control = list(trace
## = 0, : Special controls present for optimr with method snewton
```

```
proptimr(t1)
```

```
## Result  t1 ( snewton  ->  SimpHess ) calc. min. = 0  at
## 0     0     0     0
## After  2  fn evals, and  2  gr evals and  1  hessian evals
## Termination code is  0 : Small gradient norm
##
## ------------------------------------------------
```

```
t1m <- optimr(x0, fnt, grt, hesst, method="snewtonm", control=list(trace=0), fscale=3.0)
proptimr(t1m)
```

```
## Result  t1m ( snewtonm  ->  SimpHess ) calc. min. = 2.179679e-31  at
## 1.644772e-20     1.84828e-19     3.157948e-18     2.694731e-16
## After  5  fn evals, and  4  gr evals and  3  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

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

```
## Result  t1nlmo ( nlm  ->  SimpHess ) calc. min. = 8.430951e-30  at
## -2.220446e-16     2.220446e-16     -4.440892e-16     -8.881784e-16
## After  7  fn evals, and  7  gr evals and  7  hessian evals
## Termination code is  0 : nlm: Convergence indicator (code) =  1
##
## ------------------------------------------------
```

```
## BUT ... nlminb may not be using a true Newton-type method
tst <- try(t1nlminbo <- optimr(x0, fnt, grt, hess=hesst, method="nlminb",
                               fscale=3.0, control=list(trace=0)))

# Bounded
tstb <- try(t1nlminbb <- optimr(x0, fnt, grt, hess=hesst, method="nlminb",
                lower=lo, upper=up, fscale=3.0, control=list(trace=0)))
proptimr(t1nlminbb)
```

```
## Result  t1nlminbb ( nlminb  ->  SimpHess ) calc. min. = 184.5  at
## 0.5 L   1.5 L   2.5 L   3.5 L
## After  5  fn evals, and  3  gr evals and  3  hessian evals
## Termination code is  0 : relative convergence (4)
##
## ------------------------------------------------
```

```
t1smb <-  optimr(x0, fnt, grt, hess=hesst, method="snewtonm", fscale=3.0,
                 lower=lo, upper=up, control=list(trace=0))
proptimr(t1smb)
```

```
## Result  t1smb ( snewtonm  ->  SimpHess ) calc. min. = 184.5  at
## 0.5 L   1.5 L   2.5 L   3.5 L
## After  6  fn evals, and  5  gr evals and  4  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

```
# Masked
lo[1]<-x0[1]
up[1]<-x0[1]
lo[4]<-x0[4]
up[4]<-x0[4]
tstm <- try(t1nlminbm <- optimr(x0, fnt, grt, hess=hesst, method="nlminb",
                                lower=lo, upper=up, fscale=3.0, control=list(trace=0)))
proptimr(t1nlminbm)
```

```
## Result  t1nlminbm ( nlminb  ->  SimpHess ) calc. min. = 231.75  at
## 1 M   1.5 L   2.5 L   4 M
## After  4  fn evals, and  2  gr evals and  2  hessian evals
```

```
## Termination code is  0 : relative convergence (4)
##
## ------------------------------------------------
```

```
t1smm <-  optimr(x0, fnt, grt, hess=hesst, method="snewtonm", fscale=3.0,
                 lower=lo, upper=up, control=list(trace=0))
proptimr(t1smm)
```

```
## Result  t1smm ( snewtonm  ->  SimpHess ) calc. min. = 231.75  at
## 1 M   1.5 L   2.5 L   4 M
## After  4  fn evals, and  3  gr evals and  2  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

From the number of hessian evaluations, it appears `nlminb()` is also using the Hessian information. Note that the `snewton()` and `snewtonm()` functions return count information for iterations and hessian evaluations. `optimr()` also builds counts into internal scaled function, gradient and hessian functions, and these are displayed by the `proptimr()` compact output function.

The Rosenbrock function

Let us try our two Newton methods on the unconstrained Rosenbrock function and compare it to some other methods that claim to use the Hessian.

```
# RosenHess.R
require(optimx)
f <- function(x){ #Rosenbrock banana valley function
  return(100*(x[2] - x[1]*x[1])^2 + (1-x[1])^2)
}
attr(f,"fname")<-"RosenHess"
gr <- function(x){ #gradient
  return(c(-400*x[1]*(x[2] - x[1]*x[1]) - 2*(1-x[1]), 200*(x[2] - x[1]*x[1])))
}
h <- function(x) { #Hessian
  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)
t1 <- snewton(x0, fn=f, gr=gr, hess=h, control=list(trace=0))
proptimr(t1)
```

```
## Result  t1 (  ->  ) calc. min. = 0  at
## 1     1
## After  25  fn evals, and  33  gr evals and  NA  hessian evals
## Termination code is  : Small gradient norm
## Gradient:[1] 0 0
##
## ------------------------------------------------
```

```
# 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
}

t1nlmo <- optimr(x0, f, gr, hess=h, method="nlm", control=list(trace=0))
proptimr(t1nlmo)
```

```
## Result  t1nlmo ( nlm  ->  RosenHess ) calc. min. = 1.127026e-17  at
## 1     1
## After  36  fn evals, and  36  gr evals and  36  hessian 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 ( snewton  ->  RosenHess ) calc. min. = 0  at
## 1     1
## After  33  fn evals, and  25  gr evals and  24  hessian evals
## Termination code is  0 : Small gradient norm
##
## ------------------------------------------------
```

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

```
## Result  t1smo ( snewtonm  ->  RosenHess ) calc. min. = 2.946445e-27  at
## 1     1
## After  39  fn evals, and  25  gr evals and  24  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

```
## nlminb
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 ( nlminb  ->  RosenHess ) calc. min. = 0  at
## 1     1
## After  34  fn evals, and  26  gr evals and  26  hessian evals
## Termination code is  0 : X-convergence (3)
##
## ------------------------------------------------
```

```
tstnoh <- try(t1nlminbnoho <- optimr(x0, f, gr, hess=NULL, method="nlminb",
                               control=list(trace=0)))
if (class(tstnoh) == "try-error"){
  cat("try-error on attempt to run nlminb in optimr()\n")
} else { proptimr(t1nlminbnoho) }
```

```
## Result  t1nlminbnoho ( nlminb  ->  RosenHess ) calc. min. = 4.291966e-22  at
## 1     1
## After  44  fn evals, and  36  gr evals and  0  hessian 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.

```
#WoodHess.R -- Wood function example
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])
  attr(res,"fname")<-"WoodHess"
  return(res)
}
wood.g <- function(x){ #gradient
  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))
}
wood.h <- function(x){ #hessian
  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)
}
x0 <- c(-3,-1,-3,-1) # Wood standard start

require(optimx) # call methods through optimr() function
# But note that nlm default settings have lower iteration limit
#  and in 100 "iterations" do not get to solution
t1nlm <- optimr(x0, fn=wood.f, gr=wood.g, hess=wood.h, method="nlm")
proptimr(t1nlm)
```

```
## Result  t1nlm ( nlm  ->  (no_name) ) calc. min. = 1.004941e-16  at
## 1     1     1     1
## After  354  fn evals, and  354  gr evals and  354  hessian evals
## Termination code is  0 : nlm: Convergence indicator (code) =  1
##
## ------------------------------------------------
```

```
# But both optimx Newton approaches do work
wd <- optimr(x0, fn=wood.f, gr=wood.g, hess=wood.h, method="snewton")
proptimr(wd)
```

```
## Result  wd ( snewton  ->  (no_name) ) calc. min. = 1.679904e-29  at
## 1     1     1     1
## After  120  fn evals, and  69  gr evals and  69  hessian evals
## Termination code is  93 : No progress in linesearch!
##
## ------------------------------------------------
```

```
wdm <- optimr(x0, fn=wood.f, gr=wood.g, hess=wood.h, method="snewtonm")
proptimr(wdm)
```

```
## Result  wdm ( snewtonm  ->  (no_name) ) calc. min. = 6.039354e-27  at
## 1     1     1     1
## After  71  fn evals, and  49  gr evals and  48  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

```
# nlminb
t1nlminb <- optimr(x0, fn=wood.f, gr=wood.g, hess=wood.h, method="nlminb")
proptimr(t1nlminb)
```

```
## Result  t1nlminb ( nlminb  ->  (no_name) ) calc. min. = 4.373568e-28  at
## 1     1     1     1
## After  55  fn evals, and  43  gr evals and  43  hessian evals
## Termination code is  0 : X-convergence (3)
##
## ------------------------------------------------
```

```
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
}

# direct call to nlm
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
```

```
# Check that optimr gets same result with similar iteration limit of 100
t1nlmo <- optimr(x0, wood.f, wood.g, hess=wood.h, method="nlm", control=list(maxit=100))
proptimr(t1nlmo)
```

```
## Result  t1nlmo ( nlm  ->  (no_name) ) calc. min. = 7.874467  at
## -1.031607     1.074077     -0.901271     0.8239815
## After  106  fn evals, and  106  gr evals and  106  hessian evals
```

```
## Termination code is  1 : nlm: Convergence indicator (code) =  4
##
## -------------------------------------------------
```

```
print(wood.g(t1nlmo$par))
```

```
## [1]  0.007517820 -0.015797320 -0.008937001  0.015745111
```

```
# Run for allowed iteration limit in optimr 500*round(sqrt(npar+1)) = 1000
tst<-try(t1nlminbo <- optimr(x0, wood.f, wood.g, hess=wood.h, method="nlminb"))
if (class(tst) == "try-error"){
  cat("try-error on attempt to run nlminb in optimr()\n")
} else { proptimr(t1nlminbo) }
```

```
## Result  t1nlminbo ( nlminb  ->  (no_name) ) calc. min. = 4.373568e-28  at
## 1     1     1     1
## After  55  fn evals, and  43  gr evals and  43  hessian evals
## Termination code is  0 : X-convergence (3)
##
## -------------------------------------------------
```

A generalized Rosenbrock function

There are several generalizations of the Rosenbrock function (e.g., https://en.wikipedia.org/wiki/Rosenbrock_function#Multidimensional_generalisations). The following example uses the second of the Wikipedia variants.

```
# GenRoseHess.R
# 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)
}
attr(genrosa.f, "fname")<-"genrosa"

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)
# solorigs <- snewton(x0, genrosa.f, genrosa.g, genrosa.h) # WORKS OK if optimx loaded
solorig <- optimr(x0, genrosa.f, genrosa.g, genrosa.h, method="snewton", hessian=TRUE)

proptimr(solorig)
```

```
## Result  solorig ( snewton  ->  genrosa ) calc. min. = 2.972526e-28  at
## 1     1
## After  143  fn evals, and  128  gr evals and  128  hessian evals
## Termination code is  92 : No progress before linesearch!
##
## ------------------------------------------------
```

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

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

```
solorigm <- optimr(x0, genrosa.f, genrosa.g, genrosa.h, method="snewtonm", hessian=TRUE)
proptimr(solorigm)
```

```
## Result  solorigm ( snewtonm  ->  genrosa ) calc. min. = 2.012939e-27  at
## 1     1
## After  128  fn evals, and  117  gr evals and  116  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

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

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

```
# 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",
                  hessian=TRUE, gs=10)
proptimr(sol50pi)
```

```
## Result  sol50pi ( snewton  ->  genrosa ) calc. min. = 6.108742e-29  at
## 1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1
## After  145  fn evals, and  145  gr evals and  145  hessian evals
## Termination code is  92 : No progress before linesearch!
##
## ------------------------------------------------
```

```
print(eigen(sol50pi$hessian)$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
```

```
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",
                   hessian=TRUE, gs=10)
proptimr(sol50pim)
```

```
## Result  sol50pim ( snewtonm  ->  genrosa ) calc. min. = 1.707674e-21  at
## 1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1
## After  115  fn evals, and  114  gr evals and  113  hessian 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
```

```
# Bounds constraints

lo<-rep(3,50)
up<-rep(4,50)
```

```
sol50pimb <- optimr(x0, genrosa.f, genrosa.g, genrosa.h, lower=lo, upper=up, method="snewtonm",
                    hessian=TRUE, gs=10)
proptimr(sol50pimb)
```

```
## Result  sol50pimb ( snewtonm  ->  genrosa ) calc. min. = 17791.25  at
## 2.999253 -   2.999993 -   3 L   3 -   3 -   3 -   3 -   3 -   3 -   3 -   3 -   3 -   3 -   3 I
## After  30  fn evals, and  29  gr evals and  28  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

```
# approximate hessian
solom01 <- optimr(x0, genrosa.f, gr=NULL, hess="approx", method="snewtonm", hessian=TRUE)
proptimr(solom01)
```

```
## Result  solom01 ( snewtonm  ->  genrosa ) calc. min. = 8.606267e-21  at
## 1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1
## After  54  fn evals, and  34  gr evals and  33  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

```
print(eigen(solom01$hessian)$values)
```

```
##  [1] 1800.4213477 1795.6916212 1787.8294865 1776.8659723 1762.8443468
##  [6] 1745.8199474 1725.8599621 1703.0431647 1677.4596032 1649.2102453
## [11] 1618.4065792 1585.1701741 1549.6322004 1511.9329116 1472.2210914
## [16] 1430.6534661 1387.3940864 1342.6136796 1296.4889765 1249.2020132
## [21] 1200.9394133 1151.8916513 1102.2523007 1052.2172705 1001.9840317
## [26]  951.7508385  901.7159456  852.0768261  803.0293924  754.7672226
## [31]  707.4807973  661.3567475  616.5771185  573.3186509  531.7520834
## [36]  492.0414785  454.3435745  418.8071653  385.5725125  354.7707885
## [41]  326.5235555  300.9422818  278.1278960  258.1703833  241.1484269
## [46]  227.1290962  216.1675891  208.3070268  203.5783059    0.4987531
```

```
solomg1 <- optimr(x0, genrosa.f, genrosa.g, hess="approx", method="snewtonm", hessian=TRUE)
proptimr(solomg1)
```

```
## Result  solomg1 ( snewtonm  ->  genrosa ) calc. min. = 1.684908e-27  at
## 1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1
## After  54  fn evals, and  34  gr evals and  33  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

```
print(eigen(solomg1$hessian)$values)
```

```
##  [1] 1800.4213477 1795.6916212 1787.8294865 1776.8659723 1762.8443468
##  [6] 1745.8199474 1725.8599621 1703.0431647 1677.4596032 1649.2102453
## [11] 1618.4065792 1585.1701742 1549.6322004 1511.9329116 1472.2210914
## [16] 1430.6534661 1387.3940864 1342.6136796 1296.4889765 1249.2020132
## [21] 1200.9394133 1151.8916513 1102.2523007 1052.2172705 1001.9840317
## [26]  951.7508385  901.7159456  852.0768261  803.0293924  754.7672226
## [31]  707.4807973  661.3567475  616.5771185  573.3186509  531.7520834
## [36]  492.0414785  454.3435745  418.8071653  385.5725125  354.7707885
## [41]  326.5235555  300.9422818  278.1278960  258.1703833  241.1484269
## [46]  227.1290962  216.1675891  208.3070268  203.5783059    0.4987531
```

```
# Following should fail
solomrr <- try(optimr(x0, genrosa.f, gr=NULL, hess="rubbish", method="snewtonm", hessian=TRUE))
```

```
## Error in optimr(x0, genrosa.f, gr = NULL, hess = "rubbish", method = "snewtonm",  :
##   Undefined character hessian -- hess =rubbish
```

Note that the above example includes an illustration of how an approximate hessian may be invoked.

The Hobbs weed infestation problem

This problem is described in Nash (1979). It has various nasty properties. Note that one starting point causes failure of the `snewton()` optimizer.

```
# HobbHess.R
## Optimization test function HOBBS
## 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)
}
attr(hobbs.f, "fname")<- "Hobbs"

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)
}

x0 <- c(200, 50, .3)
cat("Good start for Hobbs:")
```

```
## Good start for Hobbs:
```

```
print(x0)
```

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

```
solx0 <- optimr(x0, hobbs.f, hobbs.g, hobbs.h, method="snewton", hessian=TRUE)
## Note that we exceed count limit, but have answer
proptimr(solx0)
```

```
## Result  solx0 ( snewton  ->  Hobbs ) calc. min. = 2.587277  at
## 196.1863     49.09164     0.3135697
## After  10  fn evals, and  10  gr evals and  10  hessian evals
## Termination code is  92 : No progress before linesearch!
##
## ------------------------------------------------
```

```
print(eigen(solx0$hessian)$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 <- optimr(x0, hobbs.f, hobbs.g, hobbs.h, method="snewton",
                 control=list(offset=1000.))
proptimr(solx0a)
```

```
## Result  solx0a ( snewton  ->  Hobbs ) calc. min. = 2.587277  at
## 196.1863     49.09164     0.3135697
## After  10  fn evals, and  10  gr evals and  10  hessian evals
## Termination code is  92 : No progress before linesearch!
##
## ------------------------------------------------
```

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

```
## Scaled start for Hobbs:
```

```
print(x1s)
```

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

```
solx1s <- optimr(x1s, hobbs.f, hobbs.g, hobbs.h, method="snewton", hessian=TRUE , control=list(trace=0))
proptimr(solx1s)
```

```
## Result  solx1s ( snewton  ->  Hobbs ) calc. min. = 2.587277  at
## 196.1863     49.09164     0.3135697
## After  45  fn evals, and  36  gr evals and  36  hessian evals
## Termination code is  92 : No progress before linesearch!
##
## ------------------------------------------------
```

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

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

```
solx1m <- optimr(x1s, hobbs.f, hobbs.g, hobbs.h, method="snewtonm", hessian=TRUE , control=list(trace=0))
proptimr(solx1m)
```

```
## Result  solx1m ( snewtonm  ->  Hobbs ) calc. min. = 2.587277  at
## 196.1863     49.09164     0.3135697
## After  40  fn evals, and  26  gr evals and  26  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

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

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

```
cat("Following test fails in snewton with ERROR \n
     -- Not run as function infinite.\n")
```

```
## Following test fails in snewton with ERROR
##
##      -- Not run as function infinite.
```

```
x3 <- c(1, 1, 1)
# solx3 <- try(optimr(x3, hobbs.f, hobbs.g, hobbs.h, method="snewton", control=list(trace=4)))
# if ((solx3$convergence != 0) || class(solx3) != "try-error") {
#   proptimr(solx3)
#   print(eigen(solx3$hessian)$values)
# }
# dirx3 <- try(snewton(x3, hobbs.f, hobbs.g, hobbs.h, control=list(trace=4)))
# if ((dirx3$convergence != 0) || class(dirx3) != "try-error") {
#   proptimr(dirx3)
#   print(eigen(dirx3$Hess)$values)
# }
cat("But Marquardt variant succeeds\n")
```

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

```
solx3m <- optimr(x3, hobbs.f, hobbs.g, hobbs.h, method="snewtonm",
                 hessian=TRUE, control=list(trace=0))
proptimr(solx3m)
```

```
## Result  solx3m ( snewtonm  ->  Hobbs ) calc. min. = 2.587277  at
## 196.1863     49.09164     0.3135697
## After  35  fn evals, and  24  gr evals and  23  hessian evals
## Termination code is  0 : snewtonm: Normal exit
##
## ------------------------------------------------
```

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

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

```
# we could also use nlm and nlminb and call them from optimr

solx3 <- try(optimr(x3, hobbs.f, hobbs.g, hobbs.h, method="snewton", control=list(trace=0)))
```

```
## Error in snewton(par = spar, fn = efn, gr = egr, hess = ehess, control = mcontrol) :
##   Cannot compute gradient projection
```

```
if ((class(solx3) != "try-error") && (solx3$convergence == 0)) {
  proptimr(solx3)
  print(eigen(solx3$hessian)$values)
} else cat("solx3 failed!\n")
```

```
## solx3 failed!
```

```
dirx3 <- try(snewton(x3, hobbs.f, hobbs.g, hobbs.h, control=list(trace=0)))
if ((class(dirx3) != "try-error") && (dirx3$convcode == 0)) {
  proptimr(dirx3)
  print(eigen(dirx3$Hess)$values)
} else cat("dirx3 failed!\n")
```

```
## dirx3 failed!
```

An assessment

In a number of tests, in particular using the tests in Melville (2018), the 'snewton()' approach is far from satisfactory. This is likely because the search direction computed cannot adapt to find lower function values when the Hessian is near singular. In fact, I do not include this approach in the "all methods" control in the function 'optimx::opm()'.

On the other hand, 'snewtonm()' generally works reasonably well, though it is an open question whether the gain in information in using the Hessian contributes to better solutions or better efficiency in optimization.

References

Hartley, H. O. 1961. "The Modified Gauss-Newton Method for the Fitting of Non-Linear Regression Functions by Least Squares." *Technometrics* 3: 269--280.

Melville, James. 2018. *Funconstrain: Functions for Testing Unconstrained Numerical Optimization.* https://github.com/jlmelville/funconstrain.

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