Optimization and related uses of autodiffr: Illustrations

Changcheng Li and John C. Nash

2018/7/10

Introduction

`autodiffr` is an **R** package to perform automatic differentiation of **R** functions by calling the automatic differentiation tools in the **Julia** programming language. The document **FandR** (??ref) describes how to install `autodiffr`.

Here we will illustrate how `autodiffr` can be used to provide gradients or other derivative information for use in optimization problems for which **R** is being used to attempt solutions.

Problem setup

Most methods for optimization require

- an objective function that is to be minimized or maximized. Because the mimimum of $f(x)$ is the maximum of $-f(x)$, we will only talk of minimizing functions. Though the mathematics of this are trivial, the care and attention to avoid errors when translating a maximization problem to one involving minimization require serious effort and continuous checking.
- a starting set of values for the parameters to be optimized (and perhaps also for any exogenous data and/or fixed parameters)

We need to load the `autodiffr` package and then initiate it. NOTE: This is quite slow the first time it is run. WORSE: It is always treated as a "first time" when called in knitr during the processing of a vignette from an Rmd-type file.

```
library(autodiffr)
```

```
##
## Attaching package: 'autodiffr'
```

```
## The following object is masked from 'package:stats':
##
##     deriv
```

```
ad_setup()
```

```
## Julia version 0.6.3 at location /usr/local/bin will be used.
```

```
## Loading setup script for JuliaCall...
```

```
## Finish loading setup script for JuliaCall.
```

And we can use package `numDeriv` to compare with `autodiffr`.

```
require(numDeriv)
```

```
## Loading required package: numDeriv
```

```
##
## Attaching package: 'numDeriv'
```

```
## The following objects are masked from 'package:autodiffr':
##
##     grad, hessian, jacobian
```

Test problem – ViaRes

This is simply to test how we can get the gradient of a function that is defined as the sum of squares of residuals, BUT the residuals are computed in a subsidiary function that must be called.

At July 2, 2018, this gives an error that stops knitr, so evaluation is turned off in the following example.

```
require(autodiffr)
ad_setup() # to ensure it is established

ores <- function(x){
    x # Function will be the parameters. ofn is sum of squares
}

ofn0 <- function(x){ # original ofn
   res <- ores(x) # returns a vector of residual values
   val <- as.numeric(crossprod(res)) # as.numeric because crossprod is a matrix
   val
}

ofn <- function(x){ # But autodiffr does not understand crossprod()
   res <- ores(x) # returns a vector of residual values
   val <- sum(res*res) # NOT crossprod()
   val
}
```

Note that this works with `eval=TRUE`, but Chebyquad still failing.

```
## Now try to generate the gradient function
ogr <- autodiffr::makeGradFunc(ofn)

# print(ogr) # this will be more or less meaningless link to Julia function
x0 <- c(1,2,3)
print(ofn(x0)) # should be 14
```

```
## [1] 14
```

```
print(ofn0(x0)) # should be 14
```

```
## [1] 14
```

```
ogr0<-ogr(x0) # should be 2, 4, 6
ogr0
```

```
## [1] 2 4 6
```

Test problem – Chebyquad

This problem was given prominence in the optimization literature by Fletcher (1965).

First let us define our Chebyquad function. Note that this is for the **vector** x. This version is expressed as the sum of squares of a vector of function values, which provide a nonlinear least squares problem. Note that `crossprod()` may cause difficulties as it is not written in **R**.

```
require(autodiffr)
ad_setup()

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<-zeros(x) # need to use zeros() to do initialization instead of rep() otherwise autodiffr will b
   ## This is because later on res[i] <- rr,
   ## if res is a normal R vector and rr is some Julia Number used by autodiffr,
   ## then R doesn't know what to do.
   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
}
```

Let us choose a single value for the number of parameters, and for illustration use $n = 4$.

```
## cyq.setup
n <- 4
  lower<-rep(-10.0, n)
  upper<-rep(10.0, n)
  x<-1:n
  x<-x/(n+1.0) # Initial value suggested by Fletcher
```

For safety, let us check the function and a numerical approximation to the gradient.

```
require(numDeriv)
cat("Initial parameters:")
```

```
## Initial parameters:
```

```
print(x)
```

```
## [1] 0.2 0.4 0.6 0.8
```

```
cat("Initial value of the function is ",cyq.f(x),"\n")
```

```
## Initial value of the function is  0.07118393
```

```r
gn <- numDeriv::grad(cyq.f, x) # using numDeriv
cat("Approximation to gradient at initial point:")
```

```
## Approximation to gradient at initial point:
```

```r
print(gn)
```

```
## [1]  0.6170624  0.1882112 -0.1882112 -0.6170624
```

Using a modular approach to the problem, first specifying it via **residuals** and computing the function as a sum of squares, we can also generate the gradient.

```r
# Ref: Fletcher, R. (1965) Function minimization without calculating derivatives -- a review,
#         Computer J., 8, 33-41.

# Note we do not have all components here e.g., .jsd, .h

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<- 2.0 * as.vector(rv %*% cj)
}


# check gradient function cyq.g

gajn <- cyq.g(x)
```

```r
print(gajn)
```

```
## [1]  0.6170624  0.1882112 -0.1882112 -0.6170624
```

We can now try to see if `autodiffr` matches this gradient. However, the following code gives an error in Julia.

```r
# Do not evaluate, as this fails # Now it should work
 cyq.ag <- autodiffr::makeGradFunc(cyq.f)
gaag <- cyq.ag(x)
print(gaag)
```

```
## [1]  0.6170624  0.1882112 -0.1882112 -0.6170624
```

As a workaround, we can get the Chebyquad function from the package `funconstrain`. The `funconstrain` offering does NOT require a call to the residuals, but has a single level **R** function.

```r
require(funconstrain)
```

```
## Loading required package: funconstrain
```

```r
cat("funconstrain loaded\n")
```

```
## funconstrain loaded
```

```r
cheb <- chebyquad() # Seem to need the brackets or doesn't return pieces
print(str(cheb))
```

```
## List of 4
##  $ fn:function (par)
##  $ gr:function (par)
##  $ fg:function (par)
##  $ x0:function (n = 50)
## NULL
```

```r
cyq2.f <- cheb$fn
## Note that funconstrain offers the starting value
## x0b <- cheb$x0(n=4) # Need the size of the vector
## x0b
## cyq2.f(x0b)
## same as
print(cyq2.f(x))
```

```
## [1] 0.07118393
```

```r
## Try the gradient
cyq2.ag <- autodiffr::makeGradFunc(cyq2.f) # Need autodiffr:: specified for knitr
## print(cyq2.g)
cat("Gradient at x")
```

```
## Gradient at x
```

```r
g2ag <- cyq2.ag(x)
print(g2ag)
```

```
## [1]  0.6170624  0.1882112 -0.1882112 -0.6170624
```

```r
require(microbenchmark)
```

```
## Loading required package: microbenchmark
```

```r
# NOTE: Very strange output when running 1 line at a time here in Rstudio
cat("cyq.f timing:\n")
```

```
## cyq.f timing:
```

```r
tcyq.f <- microbenchmark(cyq.f(x))
tcyq.f
```

```
## Unit: microseconds
##      expr     min      lq     mean   median       uq      max neval
##  cyq.f(x) 386.105 421.814 435.8203 427.4625 435.1135 1132.156   100
```

```r
cat("cyq2.f timing:\n")
```

```
## cyq2.f timing:
```

```r
tcyq2.f <- microbenchmark(cyq2.f(x))
summary(tcyq2.f)
```

```
##        expr   min     lq    mean median     uq    max neval
## 1 cyq2.f(x) 6.232 6.4605 6.99143  6.632 6.8575 33.252   100
```

```r
tcyq.g <- microbenchmark(cyq.g(x))
summary(tcyq.g)
```

```
##       expr     min      lq     mean   median       uq      max neval
## 1 cyq.g(x) 421.028 432.098 508.6729 461.5245 477.2615 5580.831   100
```

```r
tcyq.g <- microbenchmark(cyq.g(x))
tcyq.g
```

```
## Unit: microseconds
##      expr     min       lq     mean   median       uq      max neval
##  cyq.g(x) 421.218 436.4315 465.3071 455.4265 473.7935 1101.629   100
```

```r
cyq2.g <- cheb$gr
tcyq2.g <- microbenchmark(cyq2.g(x), unit="us" )
# microseconds
tcyq2.g
```

```
## Unit: microseconds
##       expr    min      lq     mean  median      uq    max neval
##  cyq2.g(x) 12.548 12.9735 14.03828 13.3575 14.0655 39.024   100
```

```r
# These are very slow
tcyq.ag <- microbenchmark(cyq.ag(x), unit="us" )
# microseconds
tcyq.ag
```

```
## Unit: microseconds
##       expr      min       lq     mean   median       uq      max neval
##  cyq.ag(x) 99669.59 101510.8 109016.6 105695.8 113695.7 148397.2   100
```

```r
tcyq2.ag <- microbenchmark(cyq2.ag(x), unit="us" )
# microseconds
tcyq2.ag
```

```
## Unit: microseconds
##        expr     min       lq     mean   median       uq      max neval
##  cyq2.ag(x) 21258.3 21617.71 22583.94 21999.72 22408.28 39148.11   100
```

```r
# These are quicker, but still slow
cyq.optimized_ag <- autodiffr::makeGradFunc(cyq.f, x = runif(length(x)), use_tape = TRUE)
```

```r
cyq2.optimized_ag <- autodiffr::makeGradFunc(cyq2.f, x = runif(length(x)), use_tape = TRUE)
```

```r
tcyq.optimized_ag <- microbenchmark(cyq.optimized_ag(x), unit="us" )
# microseconds
tcyq.optimized_ag
```

```
## Unit: microseconds
##                 expr     min      lq     mean   median       uq      max
##  cyq.optimized_ag(x) 897.415 903.732 4714.804 910.4105 944.7425 378929.7
##  neval
##    100
```

```r
tcyq2.optimized_ag <- microbenchmark(cyq2.optimized_ag(x), unit="us" )
# microseconds
tcyq2.optimized_ag
```

```
## Unit: microseconds
##                  expr     min       lq     mean   median      uq      max
##  cyq2.optimized_ag(x) 868.883 883.8445 2861.381 913.7315 940.917 194449.4
##  neval
##    100
```

```r
## The slowness of the optimized method is partly because the overhead of the JuliaCall and autodiffr p
## After interface functions become stable, I will try to carry on some performance optimizations,
## which is a goal of the project at last phase.
## For example, even we only have a very simple function, the timing is high because of the overhead.
foobar <- function(x) sum(x)
```

```r
foobar.ag <- autodiffr::makeGradFunc(foobar, x = runif(length(x)), use_tape = TRUE)
```

```r
tfoobar.ag <- microbenchmark(foobar.ag(x), unit="us" )
tfoobar.ag
```

```
## Unit: microseconds
##          expr     min       lq     mean   median      uq      max neval
##  foobar.ag(x) 843.367 852.1865 1037.841 859.5525 878.765 16446.05   100
```

```r
## Suppose we are dealing with input of larger size, the overhead stays roughly the same,
## so the overhead should matters not that much as in the case n = 4.
## For example, if n = 25, the diffrence of performance in ratio is not that much.
## cyq.setup
n <- 25
lower<-rep(-10.0, n)
upper<-rep(10.0, n)
x<-1:n
x<-x/(n+1.0) # Initial value suggested by Fletcher
```

```r
tcyq.g <- microbenchmark(cyq.g(x), unit = "us")
tcyq.g
```

```
## Unit: microseconds
##      expr      min       lq     mean   median      uq      max neval
##  cyq.g(x) 6407.557 6506.096 6813.755 6630.014 6858.59 12014.49   100
```

```r
tcyq2.g <- microbenchmark(cyq2.g(x), unit="us" )
# microseconds
tcyq2.g
```

```
## Unit: microseconds
##       expr    min      lq     mean median      uq     max neval
##  cyq2.g(x) 72.005 74.5475 77.59262 75.763 78.9035 107.534   100
```

```r
## The bad thing is that if the input size changes, we need to make an optimized gradient again.
```

```r
cyq.optimized_ag <- autodiffr::makeGradFunc(cyq.f, x = runif(length(x)), use_tape = TRUE)
```

```r
cyq2.optimized_ag <- autodiffr::makeGradFunc(cyq2.f, x = runif(length(x)), use_tape = TRUE)
```

```r
tcyq.optimized_ag <- microbenchmark(cyq.optimized_ag(x), unit="us" )
# microseconds
tcyq.optimized_ag
```

```
## Unit: microseconds
##                 expr     min       lq     mean   median       uq      max
##  cyq.optimized_ag(x) 11472.5 12016.28 12643.18 12306.49 12754.31 16387.49
##  neval
##    100
```

```r
tcyq2.optimized_ag <- microbenchmark(cyq2.optimized_ag(x), unit="us" )
# microseconds
tcyq2.optimized_ag
```

```
## Unit: microseconds
##                  expr     min       lq     mean   median       uq      max
##  cyq2.optimized_ag(x) 937.189 958.2585 1249.537 1002.124 1058.863 23098.52
##  neval
##    100
```

```r
## Also note it is better to check the correctness when generating optimized gradient,
all.equal(cyq.g(x), cyq.optimized_ag(x))
```

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

```r
all.equal(cyq.g(x), cyq2.optimized_ag(x))
```

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

```r
## Benchmarking times without user interface wrappers
tape1 <- reverse.grad.tape(cyq.f, runif(length(x)))
microbenchmark(reverse.grad(tape1, x), unit="us" )
```

```
## Unit: microseconds
##                    expr      min       lq    mean   median       uq
##  reverse.grad(tape1, x) 11650.72 12386.13 12887.6 12694.86 13185.99
##      max neval
##  17501.4   100
```

```r
tape2 <- reverse.grad.tape(cyq2.f, runif(length(x)))
microbenchmark(reverse.grad(tape2, x), unit="us" )
```

```
## Unit: microseconds
##                    expr   min      lq     mean  median     uq      max
##  reverse.grad(tape2, x) 929.1 934.066 965.2175 939.575 951.16 1724.895
```

```
##  neval
##    100
```

```r
## Benchmarking times without autodiffr wrappers
JuliaCall::julia_command("using ReverseDiff")
tape1 <- reverse.grad.tape(cyq.f, runif(length(x)))
microbenchmark(JuliaCall::julia_call("ReverseDiff.gradient!", tape1, x), unit="us" )
```

```
## Unit: microseconds
##                                                        expr      min
##  JuliaCall::julia_call("ReverseDiff.gradient!", tape1, x) 12135.08
##        lq     mean   median       uq      max neval
##  13220.72 13715.31 13587.86 14208.88 16231.08   100
```

```r
tape2 <- reverse.grad.tape(cyq2.f, runif(length(x)))
microbenchmark(JuliaCall::julia_call("ReverseDiff.gradient!", tape2, x), unit="us" )
```

```
## Unit: microseconds
##                                                        expr     min       lq
##  JuliaCall::julia_call("ReverseDiff.gradient!", tape2, x) 555.604 560.7235
##      mean   median       uq      max neval
##  593.0419 564.0385 594.4725 1392.169   100
```

Test problem – Hobbs weed infestation

This nonlinear estimation problem was brought to one of the authors (JN) in the mid-1970s (See Nash (1979)). It has just 12 data points and asks for the estimation of a 3-parameter logistic growth curve. The present example does not provide for scaling.

```r
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
     jjret <- jj
     attr(jjret,"gradient") <- jj
   return(jjret)
}

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)
```

```r
    return(H)
}

x0good <- c(200, 50, 0.3)
x0bad <- c(1,1,1)
f0good <- hobbs.f(x0good)
cat("Sum of squares at the GOOD starting point:",f0good,"\n")
```

```
## Sum of squares at the GOOD starting point: 158.2324
```

```r
f0bad <- hobbs.f(x0bad)
cat("Sum of squares at the BAD starting point:",f0bad,"\n")
```

```
## Sum of squares at the BAD starting point: 23520.58
```

```r
res0good <- hobbs.res(x0good)
## Residuals -- good starting point
res0good
```

```
##  [1] -0.05050236 -0.20779506 -0.26086802 -0.41247546 -0.61690702
##  [6] -1.60525418 -3.36423901 -2.43016453 -4.28734016 -5.63079076
## [11] -5.67542775 -7.44779537
```

```r
res0bad <- hobbs.res(x0bad)
## Residuals -- bad starting point
res0bad
```

```
##  [1]  -4.576941  -6.359203  -8.685426 -11.883986 -16.075693 -22.194473
##  [7] -30.443911 -37.558335 -49.156123 -61.948045 -74.995017 -90.972006
```

```r
require(autodiffr)
ad_setup()
hobbs.ag <- autodiffr::makeGradFunc(hobbs.f)
hobbsag0good <- hobbs.ag(x0good)
## Gradient by AD -- good starting point
hobbsag0good
```

```
## [1]    -17.8438     48.2491 -24559.8187
```

```r
## Compare hand coded function
hobbsggood <- hobbs.g(x0good)
hobbsggood
```

```
## [1]    -17.8438     48.2491 -24559.8187
```

```r
## Gradient by AD -- bad starting point
hobbsag0bad <- hobbs.ag(x0bad)
hobbsag0bad
```

```
## [1] -824.042084    4.764888  -11.025384
```

```r
## Compare hand coded function
hobbsgbad <- hobbs.g(x0bad)
hobbsgbad
```

```
## [1] -824.042084    4.764888  -11.025384
```

```r
## Interestingly, the magnitude of gradient elements greater for "good"
```

```r
hobbs.aj <- autodiffr::makeJacobianFunc(hobbs.res)
## Gradient by AD -- good starting point
hobbsaj0good <- hobbs.aj(x0good)
hobbsaj0good
```

```
##             [,1]       [,2]       [,3]
##  [1,] 0.02628749 -0.1023858   5.119291
##  [2,] 0.03516102 -0.1356989  13.569891
##  [3,] 0.04688566 -0.1787496  26.812437
##  [4,] 0.06226762 -0.2335615  46.712293
##  [5,] 0.08226046 -0.3019747  75.493681
##  [6,] 0.10793373 -0.3851362 115.540847
##  [7,] 0.14039380 -0.4827335 168.956738
##  [8,] 0.18063918 -0.5920347 236.813864
##  [9,] 0.22934330 -0.7069798 318.140911
## [10,] 0.28658605 -0.8178179 408.908969
## [11,] 0.35159786 -0.9119072 501.548971
## [12,] 0.42262102 -0.9760500 585.629985
```

```r
## Compare hand coded function
hobbsjgood <- hobbs.jac(x0good)
hobbsjgood
```

```
##             [,1]       [,2]       [,3]
##  [1,] 0.02628749 -0.1023858   5.119291
##  [2,] 0.03516102 -0.1356989  13.569891
##  [3,] 0.04688566 -0.1787496  26.812437
##  [4,] 0.06226762 -0.2335615  46.712293
##  [5,] 0.08226046 -0.3019747  75.493681
##  [6,] 0.10793373 -0.3851362 115.540847
##  [7,] 0.14039380 -0.4827335 168.956738
##  [8,] 0.18063918 -0.5920347 236.813864
##  [9,] 0.22934330 -0.7069798 318.140911
## [10,] 0.28658605 -0.8178179 408.908969
## [11,] 0.35159786 -0.9119072 501.548971
## [12,] 0.42262102 -0.9760500 585.629985
## attr(,"gradient")
##             [,1]       [,2]       [,3]
##  [1,] 0.02628749 -0.1023858   5.119291
##  [2,] 0.03516102 -0.1356989  13.569891
##  [3,] 0.04688566 -0.1787496  26.812437
##  [4,] 0.06226762 -0.2335615  46.712293
##  [5,] 0.08226046 -0.3019747  75.493681
##  [6,] 0.10793373 -0.3851362 115.540847
##  [7,] 0.14039380 -0.4827335 168.956738
##  [8,] 0.18063918 -0.5920347 236.813864
##  [9,] 0.22934330 -0.7069798 318.140911
## [10,] 0.28658605 -0.8178179 408.908969
## [11,] 0.35159786 -0.9119072 501.548971
## [12,] 0.42262102 -0.9760500 585.629985
```

```r
## Gradient by AD -- bad starting point
hobbsaj0bad <- hobbs.aj(x0bad)
hobbsaj0bad
```

```
##              [,1]          [,2]         [,3]
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102212e-04 6.371548e-03
##  [8,] 0.9996646 -3.352377e-04 2.681901e-03
##  [9,] 0.9998766 -1.233793e-04 1.110414e-03
## [10,] 0.9999546 -4.539581e-05 4.539581e-04
## [11,] 0.9999833 -1.670114e-05 1.837126e-04
## [12,] 0.9999939 -6.144137e-06 7.372964e-05
```

```
## Compare hand coded function
hobbsjbad <- hobbs.jac(x0bad)
hobbsjbad
```

```
##              [,1]          [,2]         [,3]
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102212e-04 6.371548e-03
##  [8,] 0.9996646 -3.352377e-04 2.681901e-03
##  [9,] 0.9998766 -1.233793e-04 1.110414e-03
## [10,] 0.9999546 -4.539581e-05 4.539581e-04
## [11,] 0.9999833 -1.670114e-05 1.837126e-04
## [12,] 0.9999939 -6.144137e-06 7.372964e-05
## attr(,"gradient")
##              [,1]          [,2]         [,3]
##  [1,] 0.7310586 -1.966119e-01 1.966119e-01
##  [2,] 0.8807971 -1.049936e-01 2.099872e-01
##  [3,] 0.9525741 -4.517666e-02 1.355300e-01
##  [4,] 0.9820138 -1.766271e-02 7.065082e-02
##  [5,] 0.9933071 -6.648057e-03 3.324028e-02
##  [6,] 0.9975274 -2.466509e-03 1.479906e-02
##  [7,] 0.9990889 -9.102212e-04 6.371548e-03
##  [8,] 0.9996646 -3.352377e-04 2.681901e-03
##  [9,] 0.9998766 -1.233793e-04 1.110414e-03
## [10,] 0.9999546 -4.539581e-05 4.539581e-04
## [11,] 0.9999833 -1.670114e-05 1.837126e-04
## [12,] 0.9999939 -6.144137e-06 7.372964e-05
```

Now let us try this in a solution of nonlinear least squares.

WARNING: Because of some compatibility issues with other **R** software, the jacobian must be available in the "gradient" attribute returned by the jacobian function. The purpose of this is to allow the function `nlsr::nlfb` to have the same name for the residual and jacobian function. This is used in generating a symbolic jacobian function in `nlsr::nlxb`. However, it can catch unwary users (including us!).

```
# try in a function
require(nlsr)
```

```
## Loading required package: nlsr
```

```
## manual
smgood <- nlfb(x0good, hobbs.res, hobbs.jac, trace=TRUE)
```

```
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 158.2324  at  = 200  = 50  = 0.3  1 / 0
## <<lamda: 4e-05  SS= 2.61779  at  = 194.3011  = 48.56497  = 0.313994  2 / 1
## <<lamda: 1.6e-05  SS= 2.587325  at  = 196.0825  = 49.07632  = 0.313616  3 / 2
## <<lamda: 6.4e-06  SS= 2.587277  at  = 196.1839  = 49.09134  = 0.313571  4 / 3
## <<lamda: 2.56e-06  SS= 2.587277  at  = 196.1862  = 49.09164  = 0.3135697  5 / 4
## <<lamda: 1.024e-06  SS= 2.587277  at  = 196.1863  = 49.09164  = 0.3135697  6 / 5
```

```
## WARNING: we need the jacobian in the "gradient" attribute
hobbs.ajx <- function(x){
     jj <- hobbs.aj(x)
     jjr <- jj
     attr(jjr, "gradient")<- jj # !!! IMPORTANT
     jjr
}

sagood <- nlfb(x0good, hobbs.res, hobbs.ajx, trace=TRUE)
```

```
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 158.2324  at  = 200  = 50  = 0.3  1 / 0
## <<lamda: 4e-05  SS= 2.61779  at  = 194.3011  = 48.56497  = 0.313994  2 / 1
## <<lamda: 1.6e-05  SS= 2.587325  at  = 196.0825  = 49.07632  = 0.313616  3 / 2
## <<lamda: 6.4e-06  SS= 2.587277  at  = 196.1839  = 49.09134  = 0.313571  4 / 3
## <<lamda: 2.56e-06  SS= 2.587277  at  = 196.1862  = 49.09164  = 0.3135697  5 / 4
## <<lamda: 1.024e-06  SS= 2.587277  at  = 196.1863  = 49.09164  = 0.3135697  6 / 5
```

Test problem – Candlestick

This function was developed by one of us to provide a simple but (for n equal 1 or 2) graphic example of a function with an infinity of solutions for $n >= 2$. The function can be seen by graphing it to have a spike in the "middle" of a dish, much like some older candlesticks or candle holders. The multiplicity of solutions should make the hessian of a solution singular. For $n = 2$, for example, the minimum lies on a circular locus at the deepest point of the "saucer".

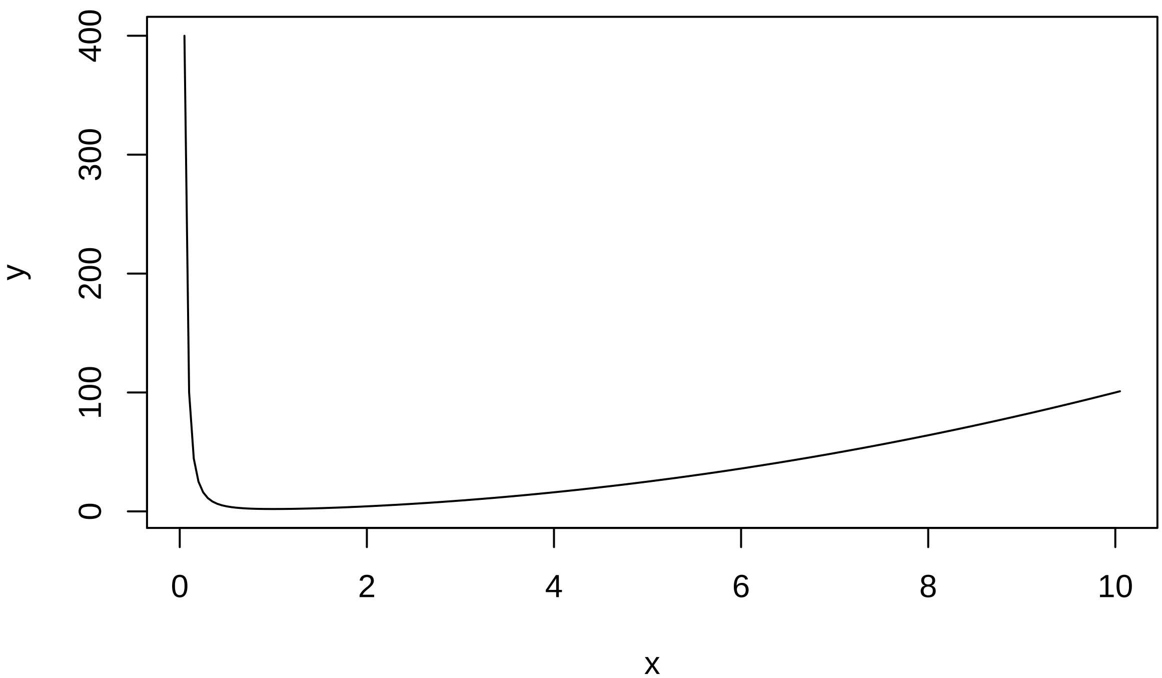
```
# candlestick function
# J C Nash 2011-2-3
cstick.f<-function(x,alpha=1){
    x<-as.vector(x)
    r2<-sum(x*x)
    f<-as.double(r2+alpha/r2)
    return(f)
}

cstick.g<-function(x,alpha=1){
    x<-as.vector(x)
    r2<-sum(x*x)
    g1<-2*x
```

```
    g2 <- (-alpha)*2*x/(r2*r2)
    g<-as.double(g1+g2)
    return(g)
}

x <- seq(-100:100)/20.0
y <- x

for (ii in 1:length(x)){
    y[ii] <- cstick.f(x[ii])
}
plot(x, y, type='l') # ?? does not plot from console??
```




```
x0 <- c(1,2)
require(optimx)
```

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

```
sdef0 <- optimr(x0, cstick.f, cstick.g, method="Rvmmin", control=list(trace=1))
```

```
## Parameter scaling:[1] 1 1
## gradient test tolerance =  6.055454e-06   fval= 5.2
##  compare to max(abs(gn-ga))/(1+abs(fval)) =  6.066115e-12
## Rvmminu -- J C Nash 2009-2015 - an R implementation of Alg 21
## Problem of size n= 2   Dot arguments:
## list()
## Initial fn= 5.2
## ig= 1   gnorm= 4.293251     1   1   5.2
## ig= 2   gnorm= 3.884638     2   2   4.468295
## *ig= 3   gnorm= 2.852834     4   3   3.087837
## *ig= 4   gnorm= 2.154552     6   4   2.520416
## ig= 5   gnorm= 3.731383     7   5   2.417429
## ig= 6   gnorm= 1.086155     8   6   2.098346
```

```
## ig= 7   gnorm= 0.504457     9   7   2.018112
## ig= 8   gnorm= 0.1371067     10   8   2.001136
## ig= 9   gnorm= 0.01377774     11   9   2.000012
## ig= 10   gnorm= 0.0003470064     12   10   2
## ig= 11   gnorm= 8.985444e-07     13   11   2
## ig= 12   gnorm= 5.846616e-11     14   12   2
## *****No acceptable point
## Converged
## Seem to be done Rvmminu
```

```
sdef0
```

```
## $par
## [1] -0.4472136 -0.8944272
##
## $value
## [1] 2
##
## $counts
## function gradient
##       19       12
##
## $convergence
## [1] 0
##
## $message
## [1] "Rvmminu appears to have converged"
```

```
xstar <- sdef0$par
gstar <- cstick.g(xstar)
cat("Gradient at proposed solution:")
```

```
## Gradient at proposed solution:
```

```
print(gstar)
```

```
## [1] -2.614686e-11 -5.229372e-11
```

```
## FIXED??
## This doesn't seem to work well??
require(autodiffr)
ad_setup()
hc <- autodiffr::makeHessianFunc(cstick.f)
hstar<-hc(xstar)
cat("Hessian at proposed solution:\n")
```

```
## Hessian at proposed solution:
```

```
print(hstar)
```

```
##      [,1] [,2]
## [1,]  1.6  3.2
## [2,]  3.2  6.4
```

```
print(eigen(hstar)$values)
```

```
## [1] 8.000000e+00 5.846612e-11
```

```
## ?? doesn't seem right
hc(x0)
```

```
##        [,1]  [,2]
## [1,] 1.984 0.128
## [2,] 0.128 2.176
```

```
require(numDeriv)
hcn0 <- numDeriv::hessian(cstick.f, x0)
hcn0
```

```
##        [,1]  [,2]
## [1,] 1.984 0.128
## [2,] 0.128 2.176
```

```
hcnstar <- numDeriv::hessian(cstick.f, xstar)
hcnstar
```

```
##      [,1] [,2]
## [1,]  1.6  3.2
## [2,]  3.2  6.4
```

```
hcnj0 <- numDeriv::jacobian(cstick.g, x0)
hcnj0
```

```
##        [,1]  [,2]
## [1,] 1.984 0.128
## [2,] 0.128 2.176
```

```
hcnjstar <- numDeriv::jacobian(cstick.g, xstar)
hcnjstar
```

```
##      [,1] [,2]
## [1,]  1.6  3.2
## [2,]  3.2  6.4
```

```
eigen(hcnstar)$values
```

```
## [1] 8.000000e+00 1.137876e-10
```

Test problem – Wood 4 parameter function

This is reported by Moré, Garbow, and Hillstrom (1980) as coming from Colville (1968). The problem in 4 parameters seems to have a false solution far from the accepted one. Is there a good description of this function and the issues it presents?

```
require(autodiffr)
ad_setup() # to ensure it is established
#Example 2: 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){
```

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

cat("Function value at x0=",wood.f(x0),"\n")
```

```
## Function value at x0= 19192
```

```r
wood.ag <- autodiffr::makeGradFunc(wood.f)
cat("Autodiffr gradient value:")
```

```
## Autodiffr gradient value:
```

```r
vwag0<-wood.ag(x0)
print(vwag0)
```

```
## [1] -12008  -2080 -10808  -1880
```

```r
cat("Manually coded:")
```

```
## Manually coded:
```

```r
vwg0 <- wood.g(x0)
print(vwg0)
```

```
## [1] -12008  -2080 -10808  -1880
```

```r
cat("Differences:\n")
```

```
## Differences:
```

```r
print(vwag0-vwg0)
```

```
## [1] 0 0 0 0
```

```r
cat("Autodiffr hessian of function value:")
```

```
## Autodiffr hessian of function value:
```

```r
wood.ah <- autodiffr::makeHessianFunc(wood.f)
vwah0 <- wood.ah(x0)
print(vwah0)
```

```
##       [,1]   [,2]  [,3]   [,4]
## [1,] 11202 1200.0     0    0.0
```

```
## [2,]  1200  220.2     0   19.8
## [3,]     0    0.0 10082 1080.0
## [4,]     0   19.8  1080  200.2
```

```
cat("Autodiffr hessian via jacobian of autodiff gradient value:")
```

```
## Autodiffr hessian via jacobian of autodiff gradient value:
```

```
wood.ahjag <- autodiffr::makeJacobianFunc(wood.ag)
vwahjag0<-wood.ahjag(x0)
print(vwahjag0)
```

```
##       [,1]   [,2]  [,3]   [,4]
## [1,] 11202 1200.0     0    0.0
## [2,]  1200  220.2     0   19.8
## [3,]     0    0.0 10082 1080.0
## [4,]     0   19.8  1080  200.2
```

```
cat("Autodiffr hessian via jacobian of manual gradient value:")
```

```
## Autodiffr hessian via jacobian of manual gradient value:
```

```
wood.ahj <- autodiffr::makeJacobianFunc(wood.g)
vwahj0 <- wood.ah(x0)
print(vwahj0)
```

```
##       [,1]   [,2]  [,3]   [,4]
## [1,] 11202 1200.0     0    0.0
## [2,]  1200  220.2     0   19.8
## [3,]     0    0.0 10082 1080.0
## [4,]     0   19.8  1080  200.2
```

```
cat("Manually coded:")
```

```
## Manually coded:
```

```
vwh0<-wood.h(x0)
print(vwh0)
```

```
##       [,1]   [,2]  [,3]   [,4]
## [1,] 11202 1200.0     0    0.0
## [2,]  1200  220.2     0   19.8
## [3,]     0    0.0 10082 1080.0
## [4,]     0   19.8  1080  200.2
```

```
cat("Differences from vwh0\n")
```

```
## Differences from vwh0
```

```
cat("vwah0\n")
```

```
## vwah0
```

```
print(vwah0-vwh0)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

```r
cat("\n")
```

```r
cat("vwahj0\n")
```

```
## vwahj0
```

```r
print(vwahj0-vwh0)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

```r
cat("\n")
```

```r
cat("vwahjag0\n")
```

```
## vwahjag0
```

```r
print(vwahjag0-vwh0)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

```r
cat("\n")
```

```r
## d <- c(1,1,1,1)
require(optimx)
meths <- c("snewton", "snewtonm", "nlm")
wdefault <- opm(x0, fn=wood.f, gr=wood.g, hess=wood.h, method=meths, control=list(trace=0))
print(wdefault)
```

```
##          p1 p2 p3 p4        value fevals gevals convergence kkt1 kkt2
## snewton   1  1  1  1 1.142616e-29    119     70          92 TRUE TRUE
## snewtonm  1  1  1  1 1.399599e-28     88     50           0 TRUE TRUE
## nlm       1  1  1  1 1.004943e-16     NA    335           0 TRUE TRUE
##          xtime
## snewton  0.008
## snewtonm 0.004
## nlm      0.004
```

```r
wagah <- opm(x0, fn=wood.f, gr=wood.ag, hess=wood.ah, method=meths, control=list(trace=0))
```

```
## Small gradient
```

```r
print(wagah)
```

```
##          p1 p2 p3 p4        value fevals gevals convergence kkt1 kkt2
## snewton   1  1  1  1 0.000000e+00    116     67           0 TRUE TRUE
## snewtonm  1  1  1  1 0.000000e+00     81     50           0 TRUE TRUE
## nlm       1  1  1  1 1.004942e-16     NA    335           0 TRUE TRUE
##           xtime
## snewton   3.868
## snewtonm  2.912
## nlm      22.724
```

```
## Timings

thand <- microbenchmark(wood.h(x0))
tad <- microbenchmark(wood.ah(x0))
print(thand)
```

```
## Unit: microseconds
##        expr   min    lq    mean median    uq    max neval
##  wood.h(x0) 4.806 5.052 5.54656 5.2405 5.612 20.994   100
```

```
print(tad)
```

```
## Unit: milliseconds
##         expr      min       lq     mean   median       uq     max neval
##  wood.ah(x0) 27.57984 28.88468 30.51616 29.42614 30.06532 59.8322   100
```

Performance issues

Optimization is, by its very nature, about improving things. Thus it is of prime interest to seek faster and better ways to optimize functions. In this section we look at some issues that may influence the speed, reliability and correctness of optimization calculations.

First, it is critical to note that **R** almost always offers several ways to accomplish the same computational result. However, the speed with which the different approaches return a result can be wildly different. (?? can JN find the 800% scale factor example??).

Second, there are many parts of the autodiffr wrapper of Julia's automatic differentiation that may use up computing cycles:

- We must translate from one programming language to another in some sense in order to call the appropriate functions in Julia based on **R** functions.
- Results must be properly structured on return to **R**.
- Hand coded derivative expressions, especially hand-optimized ones, can be expected to out-perform automatic differentiation results.

NOTE: Performance is interesting, but it is far from the complete picture. We can use results from autodiffr to validate hand-coded functions. We can get results that are efficient of human time and effort that may be otherwise unavailable. Moreover, the results of computing gradients and hessians allow us to conclude that a solution has been achieved.

A small performance comparison using `autodiffr`

```
rm(list=ls())
require(autodiffr)
autodiffr::ad_setup() # to ensure it is established

ores <- function(x){
    x # Function will be the parameters. ofn is sum of squares
}

logit <- function(x) exp(x) / (1 + exp(x))

ofn <- function(x){
```

```r
    res <- ores(x) # returns a vector of residual values
    sum(logit(res) ^ 2)
}

## Now try to generate the gradient function
ogr <- autodiffr::makeGradFunc(ofn)

system.time(ogr(runif(100)))
```

```
##    user  system elapsed
##   0.280   0.000   0.281
```

```r
system.time(ogr(runif(100)))
```

```
##    user  system elapsed
##   0.016   0.000   0.015
```

```r
ogr1 <- autodiffr::makeGradFunc(ofn, x = runif(100))

system.time(ogr1(runif(100)))
```

```
##    user  system elapsed
##   0.008   0.000   0.007
```

```r
system.time(ogr1(runif(100)))
```

```
##    user  system elapsed
##   0.008   0.000   0.008
```

```r
ogr2 <- autodiffr::makeGradFunc(ofn, x = runif(100), use_tape = TRUE)

system.time(ogr2(runif(100)))
```

```
##    user  system elapsed
##   0.140   0.000   0.142
```

```r
system.time(ogr2(runif(100)))
```

```
##    user  system elapsed
##   0.000   0.000   0.001
```

A problem with discontinuous gradient

Problems with discontinuous gradient may give gradient methods difficulty.

Here is a problem where the gradient is discontinuous, but not at the minimum.

```r
## discontin.R, a test function with discontinuous gradient

disc.f <- function(x){
   nn <- length(x)
   val <- 0.0
   for (ii in 1:nn){
       tt <- (x[ii] - ii)
       if (abs(tt) < ii) {
           ff <- tt*tt
       } else {
```

```
            ff <- abs(tt)
        }
        val <- val + ff*ff
    }
    val
}

require(optimx)
x0 <- runif(4)
x0
```

```
## [1] 0.89078635 0.47916051 0.94314744 0.05917632
```

```
sol0 <- optimr(x0, disc.f, method="nmkb")
```

```
## Warning in optimr(x0, disc.f, method = "nmkb"): Successful convergence
## Restarts for stagnation =0
```

```
sol0
```

```
## $par
## [1] 0.9978913 2.0128136 3.0128012 3.9996128
##
## $value
## [1] 5.383122e-08
##
## $convergence
## [1] 0
##
## $message
## [1] "Successful convergence"
##
## $counts
## [1] 85 NA
##
## $nitns
## [1] NA
```

```
require(autodiffr)
ad_setup()
disc.ag <- autodiffr::makeGradFunc(disc.f)
sol1 <- optimr(x0, disc.f, disc.ag, method="Rvmmin", control=list(trace=1))
```

```
## Parameter scaling:[1] 1 1 1 1
## gradient test tolerance =  6.055454e-06   fval= 264.432
##  compare to max(abs(gn-ga))/(1+abs(fval)) =  3.777694e-11
## Rvmminu -- J C Nash 2009-2015 - an R implementation of Alg 21
## Problem of size n= 4   Dot arguments:
## list()
## Initial fn= 264.432
## ig= 1   gnorm= 247.6675     1    1    264.432
## **ig= 2   gnorm= 12.27613     4    2    35.27634
## *ig= 3   gnorm= 11.32922     6    3    32.08908
## ig= 4   gnorm= 19.51258     7    4    30.11557
## *ig= 5   gnorm= 13.34165     9    5    24.49628
## ig= 6   gnorm= 33.33428   UPDATE NOT POSSIBLE: ilast, ig 1 6
```

```
##   10   6   20.04132
## *ig= 7   gnorm= 8.842687     12   7   18.74412
## **ig= 8   gnorm= 8.730798     15   8   18.2288
## ig= 9   gnorm= 4.487915     16   9   5.032437
## ig= 10   gnorm= 0.8707973     17   10   0.1467999
## ig= 11   gnorm= 0.05637104     18   11   0.00405107
## ig= 12   gnorm= 0.03706002     19   12   0.002336852
## ig= 13   gnorm= 0.01348833     20   13   0.0006334487
## ig= 14   gnorm= 0.00634206     21   14   0.0002456539
## ig= 15   gnorm= 0.002918795     22   15   9.445441e-05
## ig= 16   gnorm= 0.001628752     23   16   4.428581e-05
## ig= 17   gnorm= 0.001110674     24   17   2.472968e-05
## ig= 18   gnorm= 0.0008537438     25   18   1.603547e-05
## ig= 19   gnorm= 0.0006529062     26   19   1.087156e-05
## ig= 20   gnorm= 0.0004257771     27   20   6.226156e-06
## *ig= 21   gnorm= 0.000370275     29   21   5.295728e-06
## ig= 22   gnorm= 0.0003074603     30   22   5.009866e-06
## ig= 23   gnorm= 0.0003117494     31   23   4.579412e-06
## ig= 24   gnorm= 0.0003033878     32   24   4.476951e-06
## ig= 25   gnorm= 0.0002469098     33   25   3.687131e-06
## ig= 26   gnorm= 0.0001985193     34   26   2.806005e-06
## ig= 27   gnorm= 0.0001028793     35   27   1.1366e-06
## ig= 28   gnorm= 4.893569e-05     36   28   4.108249e-07
## ig= 29   gnorm= 2.247952e-05     37   29   1.434972e-07
## ig= 30   gnorm= 1.02367e-05     38   30   5.025503e-08
## ig= 31   gnorm= 4.538228e-06     39   31   1.706236e-08
## ig= 32   gnorm= 2.005941e-06     40   32   5.851909e-09
## ig= 33   gnorm= 1.152225e-06     41   33   2.972691e-09
## ig= 34   gnorm= 1.101338e-06     42   34   2.637216e-09
## ig= 35   gnorm= 1.110738e-06     43   35   2.605406e-09
## ig= 36   gnorm= 1.117989e-06     44   36   2.543172e-09
## ig= 37   gnorm= 1.109585e-06     45   37   2.401539e-09
## ig= 38   gnorm= 1.050001e-06     46   38   2.151045e-09
## ig= 39   gnorm= 9.52997e-07     47   39   1.899759e-09
## ig= 40   gnorm= 9.23711e-07     48   40   1.826922e-09
## ig= 41   gnorm= 9.228179e-07     49   41   1.821823e-09
## ig= 42   gnorm= 9.214369e-07     50   42   1.819489e-09
## ig= 43   gnorm= 9.168678e-07     51   43   1.815625e-09
## ig= 44   gnorm= 9.132603e-07     52   44   1.813718e-09
## ig= 45   gnorm= 9.086423e-07     53   45   1.81074e-09
## ig= 46   gnorm= 9.007073e-07     54   46   1.803488e-09
## ig= 47   gnorm= 8.855003e-07     55   47   1.783983e-09
## ig= 48   gnorm= 8.553805e-07     56   48   1.730943e-09
## ig= 49   gnorm= 7.95533e-07     57   49   1.589628e-09
## ig= 50   gnorm= 6.782673e-07     58   50   1.258063e-09
## ig= 51   gnorm= 4.720246e-07     59   51   7.318557e-10
## ig= 52   gnorm= 2.577747e-07     60   52   3.149334e-10
## ig= 53   gnorm= 1.258113e-07     61   53   1.205848e-10
## ig= 54   gnorm= 5.673724e-08     62   54   4.223402e-11
## ig= 55   gnorm= 2.62043e-08     63   55   1.670769e-11
## ig= 56   gnorm= 2.090967e-08     64   56   1.296368e-11
## ig= 57   gnorm= 1.971819e-08     65   57   1.179054e-11
## ig= 58   gnorm= 1.439377e-08     66   58   7.173373e-12
## ig= 59   gnorm= 9.262742e-09     67   59   4.156835e-12
```

```
## ig= 60   gnorm= 6.942612e-09     68   60   2.985614e-12
## ig= 61   gnorm= 5.402857e-09     69   61   2.161985e-12
## ig= 62   gnorm= 4.447603e-09     70   62   1.686036e-12
## ig= 63   gnorm= 4.141578e-09     71   63   1.516493e-12
## ig= 64   gnorm= 3.97784e-09    72   64   1.409317e-12
## ig= 65   gnorm= 3.844147e-09     73   65   1.309976e-12
## ig= 66   gnorm= 3.775019e-09     74   66   1.229767e-12
## ig= 67   gnorm= 3.77391e-09    75   67   1.200132e-12
## ig= 68   gnorm= 3.758109e-09     76   68   1.190344e-12
## ig= 69   gnorm= 3.692948e-09     77   69   1.142225e-12
## ig= 70   gnorm= 3.624243e-09     78   70   1.105634e-12
## ig= 71   gnorm= 3.360546e-09     79   71   1.003201e-12
## *ig= 72   gnorm= 3.279417e-09     81   72   9.82674e-13
## ig= 73   gnorm= 3.149905e-09     82   73   9.643588e-13
## ig= 74   gnorm= 3.215072e-09     83   74   9.521131e-13
## ig= 75   gnorm= 3.203421e-09     84   75   9.502317e-13
## ig= 76   gnorm= 3.198528e-09     85   76   9.500354e-13
## ig= 77   gnorm= 3.199201e-09     86   77   9.500134e-13
## ig= 78   gnorm= 3.200698e-09     87   78   9.499855e-13
## ig= 79   gnorm= 3.201162e-09     88   79   9.499704e-13
## ig= 80   gnorm= 3.203005e-09     89   80   9.498732e-13
## ig= 81   gnorm= 3.205116e-09     90   81   9.496754e-13
## ig= 82   gnorm= 3.208405e-09     91   82   9.491043e-13
## ig= 83   gnorm= 3.212159e-09     92   83   9.476752e-13
## ig= 84   gnorm= 3.214528e-09     93   84   9.439255e-13
## ig= 85   gnorm= 3.207566e-09     94   85   9.342785e-13
## ig= 86   gnorm= 3.163386e-09     95   86   9.081873e-13
## ig= 87   gnorm= 2.966203e-09     96   87   8.430857e-13
## ig= 88   gnorm= 2.919437e-09     97   88   8.111538e-13
## ig= 89   gnorm= 2.635285e-09     98   89   7.132601e-13
## ig= 90   gnorm= 1.055377e-09     99   90   2.161785e-13
## ig= 91   gnorm= 5.135445e-10     100   91   8.547025e-14
## ig= 92   gnorm= 2.692084e-10     101   92   3.867616e-14
## ig= 93   gnorm= 1.238153e-10     102   93   1.375787e-14
## ig= 94   gnorm= 5.282327e-11     103   94   4.377101e-15
## ig= 95   gnorm= 2.335674e-11     104   95   1.452558e-15
## ig= 96   gnorm= 1.031369e-11     105   96   4.774858e-16
## ig= 97   gnorm= 4.63775e-12    106   97   1.593293e-16
## ig= 98   gnorm= 2.119295e-12     107   98   5.378859e-17
## ig= 99   gnorm= 9.873113e-13     108   99   1.851187e-17
## ig= 100   gnorm= 4.661807e-13     109   100   6.49268e-18
## ig= 101   gnorm= 2.203452e-13     110   101   2.299234e-18
## ig= 102   gnorm= 1.025669e-13     111   102   8.040469e-19
## ig= 103   gnorm= 4.717145e-14     112   103   2.850656e-19
## ig= 104   gnorm= 2.490176e-14     113   104   1.413987e-19
## ig= 105   gnorm= 2.060521e-14     114   105   1.228037e-19
## ig= 106   gnorm= 2.039198e-14     115   106   1.223164e-19
## ig= 107   gnorm= 2.037204e-14     116   107   1.222475e-19
## ig= 108   gnorm= 2.023421e-14     117   108   1.215244e-19
## ig= 109   gnorm= 2.007859e-14     118   109   1.201898e-19
## ig= 110   gnorm= 1.979703e-14     119   110   1.16419e-19
## ig= 111   gnorm= 1.935424e-14     120   111   1.083769e-19
## ig= 112   gnorm= 1.834558e-14     121   112   9.313666e-20
## ig= 113   gnorm= 1.585084e-14     122   113   7.129666e-20
```

```
## ig= 114   gnorm= 1.13452e-14     123   114   4.520048e-20
## ig= 115   gnorm= 6.341097e-15   Seem to be done Rvmminu
```

```
sol1
```

```
## $par
## [1] 1.000005 1.999992 3.000011 3.999991
##
## $value
## [1] 2.536705e-20
##
## $counts
## function gradient
##      124      115
##
## $convergence
## [1] 2
##
## $message
## [1] "Rvmminu appears to have converged"
```

?? Do we want to try discontinuity at solution? Discontinuous function value?

Bibliography

Colville, A R. 1968. "A Comparative Study of Nonlinear Programming Codes, Rep. 320-2949." IBM New York Scientific Center.

Fletcher, R. 1965. "Function Minimization Without Calculating Derivatives – a Review." *Computer Journal* 8: 33–41.

Moré, J. J., B. S. Garbow, and K. E. Hillstrom. 1980. "ANL-80-74, User Guide for MINPACK-1." Argonne National Laboratory. http://www.mcs.anl.gov/\~{}more/ANL8074a.pdf.

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