

# 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 minimum 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)
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
```

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

## Approximation to gradient at initial point:
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.

```
# 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)
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.

```
# 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.

```
require(funconstrain)
```

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

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

```
## funconstrain loaded
```

```
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
```

```
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
```

```
## Try the gradient
```

```
cyq2.ag <- autodiffr::makeGradFunc(cyq2.f) # Need autodiffr:: specified for knitr
```

```
## print(cyq2.g)
```

```
cat("Gradient at x")
```

```
## Gradient at x
```

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

```
print(g2ag)
```

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

```
require(microbenchmark)
```

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

```
# NOTE: Very strange output when running 1 line at a time here in Rstudio
```

```
cat("cyq.f timing:\n")
```

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

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

```
tcyq.f
```

```
## Unit: microseconds
```

```
##      expr      min       lq      mean  median       uq      max neval
```

```
##  cyq.f(x) 93.483 95.7845 101.0471 96.8645 98.3855 318.578   100
```

```

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

## cyq2.f timing:
tcyq2.f <- microbenchmark(cyq2.f(x))
summary(tcyq2.f)

##      expr    min      lq    mean median      uq    max neval
## 1 cyq2.f(x) 6.402 6.578 6.97689 6.715 6.923 27.28   100

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

##      expr    min      lq    mean  median      uq    max neval
## 1 cyq.g(x) 129.995 131.969 172.6374 132.9985 134.785 3983.764   100

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

## Unit: microseconds
##      expr    min      lq    mean  median      uq    max neval
##  cyq.g(x) 128.726 131.9115 280.955 133.627 135.4235 13996.24   100

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) 10.714 11.2705 12.18925 11.758 12.531 37.518   100

# 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) 13540.38 13702.38 14571.68 13744.74 13839.89 34367.83   100

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

## Unit: microseconds
##      expr    min      lq    mean  median      uq    max neval
##  cyq2.ag(x) 2941.757 2971.6 2997.231 2984.865 2995.738 3562.696   100

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

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

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) 222.019 224.652 4131.11 225.8785 227.801 390217.1
##  neval
##      100
```

```
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) 190.854 192.581 2087.893 193.2815 194.791 189221.4
##  neval
##      100
```

*## 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)

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

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

```
## Unit: microseconds
##          expr      min      lq      mean      median      uq      max neval
##  foobar.ag(x) 172.706 174.483 179.3102 175.301 176.445 429.801   100
```

*## 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 difference 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
```

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

```
## Unit: microseconds
##          expr      min      lq      mean      median      uq      max neval
##  cyq.g(x) 6119.434 6303.066 6548.734 6388.917 6537.103 10936.32   100
```

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

```
## Unit: microseconds
##          expr      min      lq      mean      median      uq      max neval
##  cyq2.g(x) 71.597 74.62 77.5452 76.0515 79.34 114.568   100
```

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

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

```

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

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) 9716.709 10083.31 10377.6 10252.14 10451.62 15858.86
##   neval
##     100

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) 264.746 266.802 275.791 268.43 272.1005 690.891
##   neval
##     100

## Also note it is better to check the correctness when generating optimized gradient,
all.equal(cyq.g(x), cyq2.optimized_ag(x))

## [1] TRUE

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

## [1] TRUE

## 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) 9858.594 10238.48 10431.06 10395.78 10570.15
##           max neval
##  11876.49   100

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) 254.266 257.042 568.687 258.5735 262.2905 30696.16
##   neval
##     100

## 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       lq    mean  median       uq      max
##  JuliaCall::julia_call("ReverseDiff.gradient!", tape1, x) 9786.156 10091.1
##           mean  median       uq      max neval

```



```
## 10312.22 10286.03 10458.46 11486.8 100
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) 167.829 169.254
##      mean median    uq      max neval
## 173.9822 169.864 170.83 417.232 100
```

## 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.

```
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)
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

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
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

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

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

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

## [1] -17.8438 48.2491 -24559.8187

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

## [1] -824.042084 4.764888 -11.025384

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

## [1] -824.042084 4.764888 -11.025384

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

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
```

```
## 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
```

```
## 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 `nlshr::nlfb` to have the same name for the residual and jacobian function. This is used in generating a symbolic jacobian function in `nlshr::nlxb`. However, it can catch unwary users (including us!).

```
# try in a function
```

```
require(nlshr)
```

```
## Loading required package: nlshr
```

```
## 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  $n \geq 2$ . 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”.

```

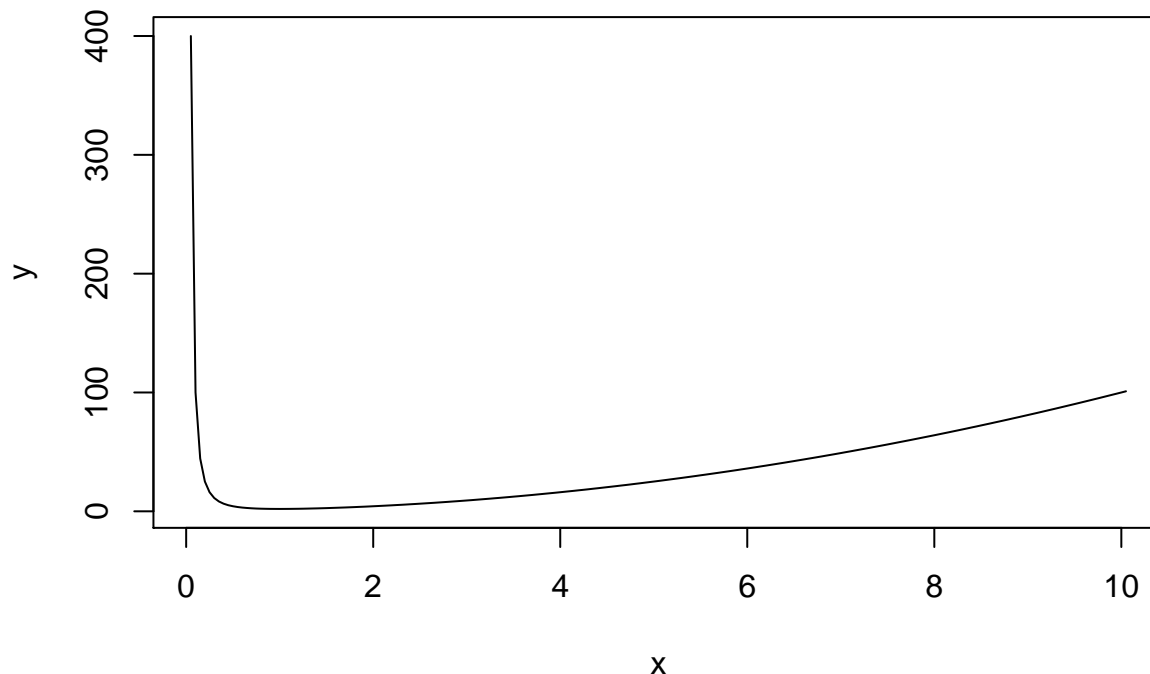
# candlestick function
# J C Nash 2011-2-3
cstick.f<-function(x,alpha=1){
  x<-as.vector(x)
  r2<-sum(x*x)
  f<-as.double(r2+alpha/r2)
  return(f)
}

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 <- optimx(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] "Rvminu 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 Hillstom (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(autodiff)
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){
  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
wood.ag <- autodiffr::makeGradFunc(wood.f)
cat("Autodiffr gradient value:")

## Autodiffr gradient value:
vwag0<-wood.ag(x0)
print(vwag0)

## [1] -12008 -2080 -10808 -1880
cat("Manually coded:")

## Manually coded:
vwg0 <- wood.g(x0)
print(vwg0)

## [1] -12008 -2080 -10808 -1880
cat("Differences:\n")

## Differences:
print(vwag0-vwg0)

## [1] 0 0 0 0
cat("Autodiffr hessian of function value:")

## Autodiffr hessian of function value:
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
```

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

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

```
## vwahj0
```

```
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
```

```

cat("\n")

cat("vwahjag0\n")

## vwahjag0
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

cat("\n")

## 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   0.004
## snewtonm  0.004
## nlm       0.008

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

## Small gradient
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   0.700
## snewtonm  0.500
## nlm       3.596

## 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) 4.733 4.9625 5.29367  5.094 5.242 22.169   100

print(tad)

## Unit: milliseconds

```

```
##      expr      min      lq      mean      median      uq      max neval
## wood.ah(x0) 5.14826 5.212993 5.894115 5.237493 5.279896 36.86989 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){
  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
##    0.224      0.000      0.225

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

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

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

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

##      user  system elapsed
##    0.004   0.000   0.002

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

##      user  system elapsed
##    0.000   0.000   0.001

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

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

##      user  system elapsed
##    0.120   0.000   0.121

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

##      user  system elapsed
##         0         0         0
```

## 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.

*## 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] 0.0542530306 0.5598794667 0.0008618799 0.8102148727
```

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

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

```
solo
```

```
## $par
## [1] 1.008340 1.984863 2.978936 3.999438
##
## $value
## [1] 2.541994e-07
##
## $convergence
## [1] 0
##
## $message
## [1] "Successful convergence"
##
## $counts
## [1] 67 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= 189.5333
## compare to max(abs(gn-ga))/(1+abs(fval)) = 1.919457e-08
## Rvmminu -- J C Nash 2009-2015 - an R implementation of Alg 21
## Problem of size n= 4 Dot arguments:
## list()
## Initial fn= 189.5333
## ig= 1 gnorm= 169.2676 1 1 189.5333
## **ig= 2 gnorm= 33.67747 4 2 20.39608
## *ig= 3 gnorm= 6.060396 6 3 9.195952
## ig= 4 gnorm= 17.09867 UPDATE NOT POSSIBLE: ilast, ig 1 4
## 7 4 8.740918
## *ig= 5 gnorm= 15.23936 9 5 5.95384
## *ig= 6 gnorm= 5.83791 11 6 1.669825
## ig= 7 gnorm= 2.731722 12 7 0.6453611
## ig= 8 gnorm= 1.126329 13 8 0.1881401
## ig= 9 gnorm= 0.5057203 14 9 0.0646646
## ig= 10 gnorm= 0.2140267 15 10 0.02054061
## ig= 11 gnorm= 0.09267295 16 11 0.006726574
## ig= 12 gnorm= 0.03977809 17 12 0.002177172
## ig= 13 gnorm= 0.01713286 18 13 0.0007078674
## ig= 14 gnorm= 0.007370383 19 14 0.0002297543
## ig= 15 gnorm= 0.003172625 20 15 7.462328e-05
## ig= 16 gnorm= 0.001365621 21 16 2.423199e-05
## ig= 17 gnorm= 0.0005879731 22 17 7.869948e-06
## ig= 18 gnorm= 0.0002532063 23 18 2.556101e-06
## ig= 19 gnorm= 0.0001090729 24 19 8.303301e-07
## ig= 20 gnorm= 4.699935e-05 25 20 2.697784e-07
```

|            |                     |    |    |              |
|------------|---------------------|----|----|--------------|
| ## ig= 21  | gnorm= 2.025843e-05 | 26 | 21 | 8.767518e-08 |
| ## ig= 22  | gnorm= 8.734615e-06 | 27 | 22 | 2.850192e-08 |
| ## ig= 23  | gnorm= 3.76679e-06  | 28 | 23 | 9.268465e-09 |
| ## ig= 24  | gnorm= 1.624637e-06 | 29 | 24 | 3.015706e-09 |
| ## ig= 25  | gnorm= 7.00966e-07  | 30 | 25 | 9.837528e-10 |
| ## ig= 26  | gnorm= 3.031478e-07 | 31 | 26 | 3.250138e-10 |
| ## ig= 27  | gnorm= 1.329922e-07 | 32 | 27 | 1.131921e-10 |
| ## ig= 28  | gnorm= 6.313858e-08 | 33 | 28 | 4.665181e-11 |
| ## ig= 29  | gnorm= 3.951643e-08 | 34 | 29 | 2.698773e-11 |
| ## ig= 30  | gnorm= 3.499442e-08 | 35 | 30 | 2.19804e-11  |
| ## ig= 31  | gnorm= 3.510645e-08 | 36 | 31 | 2.106839e-11 |
| ## ig= 32  | gnorm= 3.5441e-08   | 37 | 32 | 2.098352e-11 |
| ## ig= 33  | gnorm= 3.551736e-08 | 38 | 33 | 2.097992e-11 |
| ## ig= 34  | gnorm= 3.554217e-08 | 39 | 34 | 2.097814e-11 |
| ## ig= 35  | gnorm= 3.561491e-08 | 40 | 35 | 2.09694e-11  |
| ## ig= 36  | gnorm= 3.569862e-08 | 41 | 36 | 2.095068e-11 |
| ## ig= 37  | gnorm= 3.581173e-08 | 42 | 37 | 2.089825e-11 |
| ## ig= 38  | gnorm= 3.589971e-08 | 43 | 38 | 2.076742e-11 |
| ## ig= 39  | gnorm= 3.580238e-08 | 44 | 39 | 2.043383e-11 |
| ## ig= 40  | gnorm= 3.502848e-08 | 45 | 40 | 1.962605e-11 |
| ## ig= 41  | gnorm= 3.242534e-08 | 46 | 41 | 1.782057e-11 |
| ## ig= 42  | gnorm= 2.665651e-08 | 47 | 42 | 1.456175e-11 |
| ## ig= 43  | gnorm= 2.021525e-08 | 48 | 43 | 1.128328e-11 |
| ## ig= 44  | gnorm= 1.734994e-08 | 49 | 44 | 9.532706e-12 |
| ## ig= 45  | gnorm= 1.140619e-08 | 50 | 45 | 5.259768e-12 |
| ## ig= 46  | gnorm= 7.175042e-09 | 51 | 46 | 2.608782e-12 |
| ## ig= 47  | gnorm= 4.253123e-09 | 52 | 47 | 1.268626e-12 |
| ## ig= 48  | gnorm= 3.784869e-09 | 53 | 48 | 1.192451e-12 |
| ## ig= 49  | gnorm= 3.788683e-09 | 54 | 49 | 1.146095e-12 |
| ## ig= 50  | gnorm= 3.768147e-09 | 55 | 50 | 1.125713e-12 |
| ## ig= 51  | gnorm= 3.574038e-09 | 56 | 51 | 1.028194e-12 |
| ## ig= 52  | gnorm= 3.191002e-09 | 57 | 52 | 9.328436e-13 |
| ## *ig= 53 | gnorm= 3.201541e-09 | 59 | 53 | 9.272185e-13 |
| ## ig= 54  | gnorm= 3.187629e-09 | 60 | 54 | 9.235154e-13 |
| ## ig= 55  | gnorm= 3.177935e-09 | 61 | 55 | 9.23197e-13  |
| ## ig= 56  | gnorm= 3.175612e-09 | 62 | 56 | 9.231802e-13 |
| ## ig= 57  | gnorm= 3.173384e-09 | 63 | 57 | 9.23163e-13  |
| ## ig= 58  | gnorm= 3.171192e-09 | 64 | 58 | 9.23135e-13  |
| ## ig= 59  | gnorm= 3.166738e-09 | 65 | 59 | 9.230463e-13 |
| ## ig= 60  | gnorm= 3.159736e-09 | 66 | 60 | 9.228287e-13 |
| ## ig= 61  | gnorm= 3.14726e-09  | 67 | 61 | 9.222387e-13 |
| ## ig= 62  | gnorm= 3.12523e-09  | 68 | 62 | 9.206931e-13 |
| ## ig= 63  | gnorm= 3.084237e-09 | 69 | 63 | 9.165612e-13 |
| ## ig= 64  | gnorm= 3.005612e-09 | 70 | 64 | 9.055133e-13 |
| ## ig= 65  | gnorm= 2.85169e-09  | 71 | 65 | 8.759063e-13 |
| ## ig= 66  | gnorm= 2.55967e-09  | 72 | 66 | 7.989264e-13 |
| ## ig= 67  | gnorm= 2.060204e-09 | 73 | 67 | 6.240576e-13 |
| ## ig= 68  | gnorm= 1.379845e-09 | 74 | 68 | 3.672801e-13 |
| ## ig= 69  | gnorm= 9.048241e-10 | 75 | 69 | 2.027668e-13 |
| ## ig= 70  | gnorm= 7.625919e-10 | 76 | 70 | 1.448382e-13 |
| ## ig= 71  | gnorm= 7.478291e-10 | 77 | 71 | 1.27766e-13  |
| ## ig= 72  | gnorm= 7.534194e-10 | 78 | 72 | 1.229271e-13 |
| ## ig= 73  | gnorm= 7.55138e-10  | 79 | 73 | 1.209801e-13 |
| ## ig= 74  | gnorm= 7.508184e-10 | 80 | 74 | 1.194273e-13 |



|            |                     |                          |     |              |
|------------|---------------------|--------------------------|-----|--------------|
| ## ig= 75  | gnorm= 7.374331e-10 | 81                       | 75  | 1.172284e-13 |
| ## *ig= 76 | gnorm= 7.311185e-10 | 83                       | 76  | 1.165494e-13 |
| ## ig= 77  | gnorm= 7.221414e-10 | 84                       | 77  | 1.164307e-13 |
| ## ig= 78  | gnorm= 7.228852e-10 | 85                       | 78  | 1.16302e-13  |
| ## ig= 79  | gnorm= 7.213428e-10 | 86                       | 79  | 1.1629e-13   |
| ## ig= 80  | gnorm= 7.206294e-10 | 87                       | 80  | 1.162875e-13 |
| ## ig= 81  | gnorm= 7.203111e-10 | 88                       | 81  | 1.162857e-13 |
| ## ig= 82  | gnorm= 7.194324e-10 | 89                       | 82  | 1.162784e-13 |
| ## ig= 83  | gnorm= 7.188431e-10 | 90                       | 83  | 1.162691e-13 |
| ## ig= 84  | gnorm= 7.188473e-10 | 91                       | 84  | 1.162589e-13 |
| ## ig= 85  | gnorm= 7.194401e-10 | 92                       | 85  | 1.162539e-13 |
| ## ig= 86  | gnorm= 7.199026e-10 | 93                       | 86  | 1.162522e-13 |
| ## ig= 87  | gnorm= 7.202452e-10 | 94                       | 87  | 1.162506e-13 |
| ## ig= 88  | gnorm= 7.208891e-10 | 95                       | 88  | 1.162459e-13 |
| ## ig= 89  | gnorm= 7.218331e-10 | 96                       | 89  | 1.162344e-13 |
| ## ig= 90  | gnorm= 7.23345e-10  | 97                       | 90  | 1.162037e-13 |
| ## ig= 91  | gnorm= 7.256055e-10 | 98                       | 91  | 1.161247e-13 |
| ## ig= 92  | gnorm= 7.288454e-10 | 99                       | 92  | 1.159188e-13 |
| ## ig= 93  | gnorm= 7.329081e-10 | 100                      | 93  | 1.153867e-13 |
| ## ig= 94  | gnorm= 7.363282e-10 | 101                      | 94  | 1.140042e-13 |
| ## ig= 95  | gnorm= 7.335703e-10 | 102                      | 95  | 1.10721e-13  |
| ## ig= 96  | gnorm= 7.144437e-10 | 103                      | 96  | 1.061147e-13 |
| ## ig= 97  | gnorm= 5.311756e-10 | 104                      | 97  | 7.086259e-14 |
| ## ig= 98  | gnorm= 4.216958e-10 | 105                      | 98  | 5.21205e-14  |
| ## ig= 99  | gnorm= 1.571911e-10 | 106                      | 99  | 1.470393e-14 |
| ## ig= 100 | gnorm= 1.160766e-10 | 107                      | 100 | 1.104313e-14 |
| ## ig= 101 | gnorm= 8.475245e-11 | 108                      | 101 | 7.272104e-15 |
| ## ig= 102 | gnorm= 3.485242e-11 | 109                      | 102 | 2.255951e-15 |
| ## ig= 103 | gnorm= 2.04461e-11  | 110                      | 103 | 1.254426e-15 |
| ## ig= 104 | gnorm= 1.452122e-11 | 111                      | 104 | 7.941324e-16 |
| ## ig= 105 | gnorm= 5.069651e-12 | 112                      | 105 | 1.955047e-16 |
| ## ig= 106 | gnorm= 2.37138e-12  | 113                      | 106 | 7.181712e-17 |
| ## ig= 107 | gnorm= 1.032878e-12 | 114                      | 107 | 2.482385e-17 |
| ## ig= 108 | gnorm= 5.428749e-13 | 115                      | 108 | 1.093899e-17 |
| ## ig= 109 | gnorm= 3.771728e-13 | 116                      | 109 | 6.159466e-18 |
| ## ig= 110 | gnorm= 3.158461e-13 | 117                      | 110 | 4.167018e-18 |
| ## ig= 111 | gnorm= 2.661755e-13 | 118                      | 111 | 2.932157e-18 |
| ## ig= 112 | gnorm= 2.039444e-13 | 119                      | 112 | 1.918821e-18 |
| ## ig= 113 | gnorm= 1.302338e-13 | 120                      | 113 | 1.039678e-18 |
| ## ig= 114 | gnorm= 6.172679e-14 | 121                      | 114 | 4.316495e-19 |
| ## ig= 115 | gnorm= 5.121933e-14 | 122                      | 115 | 3.92124e-19  |
| ## ig= 116 | gnorm= 4.635978e-14 | 123                      | 116 | 3.408346e-19 |
| ## ig= 117 | gnorm= 1.465878e-14 | 124                      | 117 | 7.368095e-20 |
| ## ig= 118 | gnorm= 7.193312e-15 | Seem to be done Rvmmminu |     |              |

sol1

```
## $par
## [1] 1.000012 1.999992 2.999997 4.000009
##
## $value
## [1] 2.852338e-20
##
## $counts
## function gradient
```

```
##      125      118
##
## $convergence
## [1] 2
##
## $message
## [1] "Rvminu appears to have converged"
?? Do we want to try discontinuity at solution? Discontinuous function value?
```

## Beware: R constructs that do not work well in Julia and autodiff

```
require(autodiff)
f1 <- function(x){ sum(x^2)^2}
f2 <- function(x){as.numeric(crossprod(crossprod(x)))}
f3 <- function(x){ as.numeric((t(x) %*% x)^2)}
x0 <- (1:4)
print(x0)

## [1] 1 2 3 4
print(f1(x0))

## [1] 900
print(f2(x0))

## [1] 900
print(f3(x0))

## [1] 900
## Now try gradients
g1 <- makeGradFunc(f1)
print(g1(x0))

## [1] 120 240 360 480
## But these do not work
g2 <- makeGradFunc(f2)
print(try(g2(x0)))

## [1] "Error : Error happens in Julia.\nREvalError: \n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError: Error happens in Julia.
## REvalError: >
g3 <- makeGradFunc(f3)
print(try(g3(x0)))

## [1] "Error : Error happens in Julia.\nREvalError: \n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError: Error happens in Julia.
```

```

## REvalError: >
## ad_variant gets a little further, but still fails
f2v <- ad_variant(f2)
print(f2v(x0))

## [1] 900
f3v <- ad_variant(f3)
print(f3v(x0))

## [1] 900
g2v <- makeGradFunc(f2v)
print(try(g2v(x0)))

## [1] "Error : Error happens in Julia.\nREvalError: \n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError: Error happens in Julia.
## REvalError: >
g3v <- makeGradFunc(f3v)
print(try(g3v(x0)))

## [1] "Error : Error happens in Julia.\nREvalError: \n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError: Error happens in Julia.
## REvalError: >

```

## 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. Hillstom. 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.