

Optimization and related uses of autodiff: Illustrations

Changcheng Li and John C. Nash

2018/7/10

Introduction

`autodiff` 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 `autodiff`.

Here we will illustrate how `autodiff` 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 `autodiff` 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(autodiff)
```

```
##
## Attaching package: 'autodiff'
## 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 `autodiff`.

```
require(numDeriv)
```

```
## Loading required package: numDeriv
##
## Attaching package: 'numDeriv'
## The following objects are masked from 'package:autodiff':
##
##      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
```

```
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) 380.779 383.4715 398.9616 385.072 389.996 1037.833   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)  6.26  6.4635  6.8059  6.5585  6.744  24.698   100
```

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

```
##      expr      min       lq      mean  median      uq      max neval
## 1 cyq.g(x) 419.598 425.064 478.6196 427.9055 447.8875 4448.845   100
```

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

```
## Unit: microseconds
```

```
##      expr      min       lq      mean  median      uq      max neval
##  cyq.g(x) 419.968 423.9475 455.2405 428.784 470.145 1172.294   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) 10.911 11.234 11.83836 11.4375 11.9205 32.099   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) 98190.98 100321.9 106327.3 101798.4 112054.4 145310.2   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) 20972.58 21130.04 21949.57 21250.32 21713.71 36947.36   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) 888.708 895.455 4627.811 899.3065 911.369 372068.7
##    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) 862.12 869.9045 2763.294 874.826 886.5075 188133.5
##    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) 841.894 848.1965 1029.177 852.7085 869.5875 16453.97   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) 6529.1 6742.062 7203.358 7138.8 7455.367 11551.83   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) 81.562 83.341 87.05446 84.9925 88.367 120.397   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) 11405.19 11959.8 12322.47 12218.73 12585.19 13796.37
##    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) 931.363 942.799 1173.529 954.726 994.197 19611.28
##    neval
##      100

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

## [1] TRUE
all.equal(cyq2.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) 11574.73 12190.92 12688.74 12508.22 12908.39
##      max neval
## 15941.84   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
##  reverse.grad(tape2, x) 925.894 930.7715 963.1801 936.2475 952.7505

```



```

##          max neval
## 1643.158    100
## Benchmarking times without autodiff 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) 10841.77
##          lq      mean    median      uq      max neval
## 11263.78 11485.47 11458.06 11683.11 13371.03    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) 544.814 549.0625
##          mean    median      uq      max neval
## 565.0156 550.6655 554.0205 1209.66    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")<- jjr # !!! 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
```

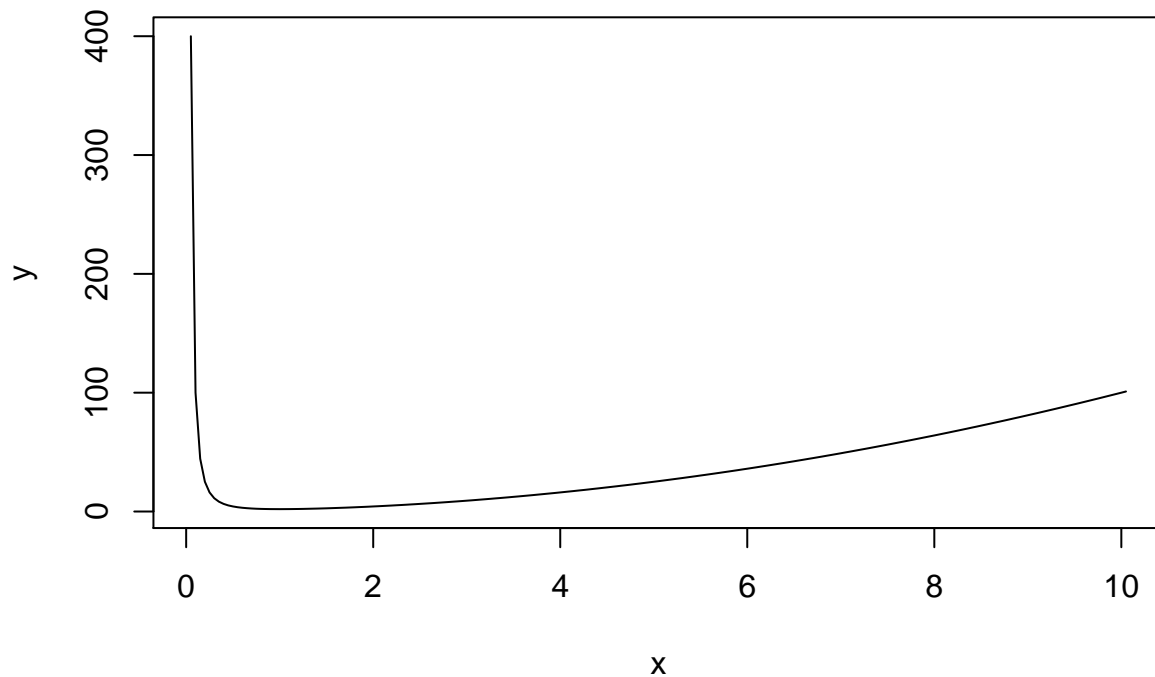
```

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 Rvmmminu
```

```
sdef0
```

```
## $par
## [1] -0.4472136 -0.8944272
##
## $value
## [1] 2
##
## $counts
## function gradient
##      19      12
##
## $convergence
## [1] 0
##
## $message
## [1] "Rvmmminu 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  0.008
## 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  3.684
## snewtonm 2.748
## nlm      20.996

```

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
## 0.232   0.000   0.232

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

##    user  system elapsed
## 0.012   0.000   0.014
```

```

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

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

##      user  system elapsed
##    0.008   0.000   0.006

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

##      user  system elapsed
##    0.008   0.000   0.007

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

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

##      user  system elapsed
##    0.120   0.000   0.123

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

##      user  system elapsed
##    0.000   0.000   0.002

```

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] 0.05623257 0.49678922 0.17867686 0.72844112

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

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

```

```
solo
```

```
## $par
## [1] 1.011852 1.989940 3.005981 4.011551
##
## $value
## [1] 4.905641e-08
##
## $convergence
## [1] 0
##
## $message
## [1] "Successful convergence"
##
## $counts
## [1] 80 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= 183.8151
## compare to max(abs(gn-ga))/(1+abs(fval)) = 9.095435e-11
## Rvmminu -- J C Nash 2009-2015 - an R implementation of Alg 21
## Problem of size n= 4 Dot arguments:
## list()
## Initial fn= 183.8151
## ig= 1 gnorm= 166.9819 1 1 183.8151
## **ig= 2 gnorm= 50.86141 4 2 31.15439
## *ig= 3 gnorm= 8.634007 6 3 18.64752
## *ig= 4 gnorm= 7.852362 8 4 15.43309
## ig= 5 gnorm= 6.202497 9 5 1.874372
## ig= 6 gnorm= 3.608616 10 6 1.041764
## ig= 7 gnorm= 2.30624 11 7 0.528743
## ig= 8 gnorm= 0.8602945 12 8 0.146082
## ig= 9 gnorm= 0.4037773 13 9 0.05687725
## ig= 10 gnorm= 0.1873 14 10 0.02190417
## ig= 11 gnorm= 0.0824998 15 11 0.007950162
## ig= 12 gnorm= 0.04092803 16 12 0.003354839
## ig= 13 gnorm= 0.02087654 17 13 0.00138061
## ig= 14 gnorm= 0.01040388 18 14 0.0005226148
## ig= 15 gnorm= 0.005158195 19 15 0.0001915578
## ig= 16 gnorm= 0.002567062 20 16 6.998395e-05
## ig= 17 gnorm= 0.00128117 21 17 2.582983e-05
## ig= 18 gnorm= 0.0006326072 22 18 9.577861e-06
## ig= 19 gnorm= 0.0003021778 23 19 3.460525e-06
## ig= 20 gnorm= 0.000137744 24 20 1.194642e-06
## ig= 21 gnorm= 6.974423e-05 25 21 5.39633e-07
## ig= 22 gnorm= 5.932248e-05 26 22 4.646484e-07
## ig= 23 gnorm= 5.880828e-05 27 23 4.594861e-07
```

## ig= 24	gnorm= 5.756863e-05	28	24	4.43656e-07
## ig= 25	gnorm= 5.509133e-05	29	25	4.077777e-07
## ig= 26	gnorm= 4.895528e-05	30	26	3.279426e-07
## ig= 27	gnorm= 3.614025e-05	31	27	2.046415e-07
## ig= 28	gnorm= 2.141538e-05	32	28	1.028975e-07
## ig= 29	gnorm= 1.305546e-05	33	29	5.903955e-08
## ig= 30	gnorm= 1.076027e-05	34	30	4.690418e-08
## ig= 31	gnorm= 1.069358e-05	35	31	4.514861e-08
## ig= 32	gnorm= 1.071694e-05	36	32	4.492917e-08
## ig= 33	gnorm= 1.068125e-05	37	33	4.459123e-08
## ig= 34	gnorm= 1.050339e-05	38	34	4.416124e-08
## ig= 35	gnorm= 1.025974e-05	39	35	4.375967e-08
## ig= 36	gnorm= 1.011209e-05	40	36	4.355825e-08
## ig= 37	gnorm= 1.010324e-05	41	37	4.353355e-08
## ig= 38	gnorm= 9.44713e-06	42	38	4.108223e-08
## ig= 39	gnorm= 8.705594e-06	43	39	3.721528e-08
## ig= 40	gnorm= 7.064965e-06	44	40	2.731678e-08
## ig= 41	gnorm= 4.623716e-06	45	41	1.452393e-08
## ig= 42	gnorm= 2.415459e-06	46	42	5.792943e-09
## ig= 43	gnorm= 1.164394e-06	47	43	2.145171e-09
## ig= 44	gnorm= 5.485104e-07	48	44	8.286538e-10
## ig= 45	gnorm= 3.28126e-07	49	45	4.703588e-10
## ig= 46	gnorm= 3.047308e-07	50	46	4.230861e-10
## ig= 47	gnorm= 3.06501e-07	51	47	4.21574e-10
## ig= 48	gnorm= 3.068087e-07	52	48	4.213081e-10
## ig= 49	gnorm= 3.083339e-07	53	49	4.193984e-10
## ig= 50	gnorm= 3.093962e-07	54	50	4.166554e-10
## ig= 51	gnorm= 3.085303e-07	55	51	4.112362e-10
## ig= 52	gnorm= 3.014456e-07	56	52	3.988589e-10
## *ig= 53	gnorm= 2.956564e-07	58	53	3.944952e-10
## ig= 54	gnorm= 2.929372e-07	59	54	3.922639e-10
## ig= 55	gnorm= 2.881376e-07	60	55	3.90585e-10
## ig= 56	gnorm= 2.881713e-07	61	56	3.901746e-10
## ig= 57	gnorm= 2.899448e-07	62	57	3.888229e-10
## ig= 58	gnorm= 2.905285e-07	63	58	3.88667e-10
## ig= 59	gnorm= 2.913638e-07	64	59	3.883233e-10
## ig= 60	gnorm= 2.923405e-07	65	60	3.875443e-10
## ig= 61	gnorm= 2.933965e-07	66	61	3.854667e-10
## ig= 62	gnorm= 2.934895e-07	67	62	3.80278e-10
## ig= 63	gnorm= 2.893251e-07	68	63	3.671618e-10
## ig= 64	gnorm= 2.705693e-07	69	64	3.318246e-10
## *ig= 65	gnorm= 2.513028e-07	71	65	3.014558e-10
## *ig= 66	gnorm= 2.395125e-07	73	66	2.882406e-10
## ig= 67	gnorm= 2.155007e-07	74	67	2.609173e-10
## ig= 68	gnorm= 1.480082e-07	75	68	1.683803e-10
## ig= 69	gnorm= 5.547375e-08	76	69	4.634057e-11
## ig= 70	gnorm= 2.513959e-08	77	70	1.627365e-11
## ig= 71	gnorm= 1.06134e-08	78	71	5.188397e-12
## ig= 72	gnorm= 4.625931e-09	79	72	1.72025e-12
## ig= 73	gnorm= 2.009678e-09	80	73	5.648867e-13
## ig= 74	gnorm= 8.886868e-10	81	74	1.880918e-13
## ig= 75	gnorm= 4.014871e-10	82	75	6.340753e-14
## ig= 76	gnorm= 1.877443e-10	83	76	2.189476e-14
## ig= 77	gnorm= 9.142242e-11	84	77	7.818376e-15


```

## ig= 78    gnorm= 4.609882e-11    85    78    2.909058e-15
## ig= 79    gnorm= 2.349838e-11    86    79    1.117073e-15
## ig= 80    gnorm= 1.163638e-11    87    80    4.235039e-16
## ig= 81    gnorm= 5.390796e-12    88    81    1.506268e-16
## ig= 82    gnorm= 2.745893e-12    89    82    7.098121e-17
## ig= 83    gnorm= 2.44704e-12    90    83    6.521574e-17
## ig= 84    gnorm= 2.445677e-12    91    84    6.51845e-17
## ig= 85    gnorm= 2.401705e-12    92    85    6.382084e-17
## ig= 86    gnorm= 2.346367e-12    93    86    6.149504e-17
## ig= 87    gnorm= 2.205291e-12    94    87    5.480209e-17
## ig= 88    gnorm= 1.889808e-12    95    88    4.16832e-17
## ig= 89    gnorm= 1.281962e-12    96    89    2.303025e-17
## ig= 90    gnorm= 6.871444e-13    97    90    9.668389e-18
## ig= 91    gnorm= 3.264181e-13    98    91    3.583068e-18
## ig= 92    gnorm= 1.423768e-13    99    92    1.21524e-18
## ig= 93    gnorm= 6.903695e-14    100   93    5.487897e-19
## ig= 94    gnorm= 5.745451e-14    101   94    4.635661e-19
## ig= 95    gnorm= 5.234113e-14    102   95    4.107397e-19
## ig= 96    gnorm= 3.719458e-14    103   96    2.477308e-19
## ig= 97    gnorm= 2.112858e-14    104   97    1.112968e-19
## ig= 98    gnorm= 1.103869e-14    105   98    4.990272e-20
## ig= 99    gnorm= 7.846098e-15    Seem to be done Rvmmminu

```

```
sol1
```

```

## $par
## [1] 0.9999892 2.0000050 2.9999921 4.0000112
##
## $value
## [1] 3.398664e-20
##
## $counts
## function gradient
##      106      99
##
## $convergence
## [1] 2
##
## $message
## [1] "Rvmmminu 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. 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.