

# Optimization and related uses of autodiff: Illustrations

*Changcheng Li and John C. Nash*

*2018/7/1*

## 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/share/julia-d55cad350/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
  print(res)
  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
  # print(res)
  val <- sum(res*res) # as.numeric because crossprod is a matrix
  val
}

## Now try to generate the gradient function
ogr <- autodiffr::grad(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
print(ofn0(x0)) # should be 14
ogr0<-ogr(x0) # should be 2, 4, 6
ogr0
```

## Test problem – Chebyquad

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

Here are some extra pieces of JN's cyq.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<- as.vector(2.0* rv %*% cj)
}

```

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.

```

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<-rep(0,n) # initialize
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

```

We can now try to see if `autodiffr` matches this gradient.

We can get the Chebyquad function from the package `funconstrain` as well as from JN's function collection. Moreover, 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$fn
x0b <- cheb$x0(n=4) # Need the size of the vector
x0b

## [1] 0.2 0.4 0.6 0.8

cyq2.f(x0b)

## [1] 0.07118393

Split chunks to narrow down error

## Try the gradient
cyq2.g <- autodiffr::grad(cyq2.f) # Works in console, NOT in knitr without autodiffr specified ??

Isolate the troublesome line.

print(cyq2.g)

## function (x)
## D[[mode]](target, scalar2vector(x))
## <bytecode: 0x3ecc870>
## <environment: 0x7f25c08>

cat("Gradient at x0b")

## Gradient at x0b

print(cyq2.g(x0b))

## [1] 0.6170624 0.1882112 -0.1882112 -0.6170624

cyq.ag <- autodiffr::grad(cyq2.f) # should return a function
print(cyq.ag)

## function (x)
## D[[mode]](target, scalar2vector(x))
## <bytecode: 0x3ecc870>
## <environment: 0x8b45b50>

cat("Result of grad(cyq.f):\n")

## Result of grad(cyq.f):
# print(cyq.ag)
cat("Evaluated at x0:")

## Evaluated at x0:

cyqag0b <- cyq.ag(x0b)
print(cyqag0b)

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

Chebyquad Questions: ec - How fast is the calculation? Try different ways and produce a table.

- Why is JN's cyq.ag NOT working? Does it have something to do with calling cyq.res to compute cyq.f? NO. crossprod()

## 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)
cat("Residuals -- good starting point\n")

## 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)
cat("Residuals -- bad starting point\n")

## 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::grad(hobbs.f)
hobbsag0good <- hobbs.ag(x0good)
## Gradient by AD -- good starting point
hobbsag0good

## [1] -17.8438 48.2491 -24559.8187
cat("Compare hand coded function:\n")

## 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
cat("Compare hand coded function:\n")

## 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::jacobian(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
```

```
cat("Compare hand coded function:\n")
```

```
## 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
```

```
cat("Compare hand coded function:\n")
```

```
## 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 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
smggood <- 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
  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

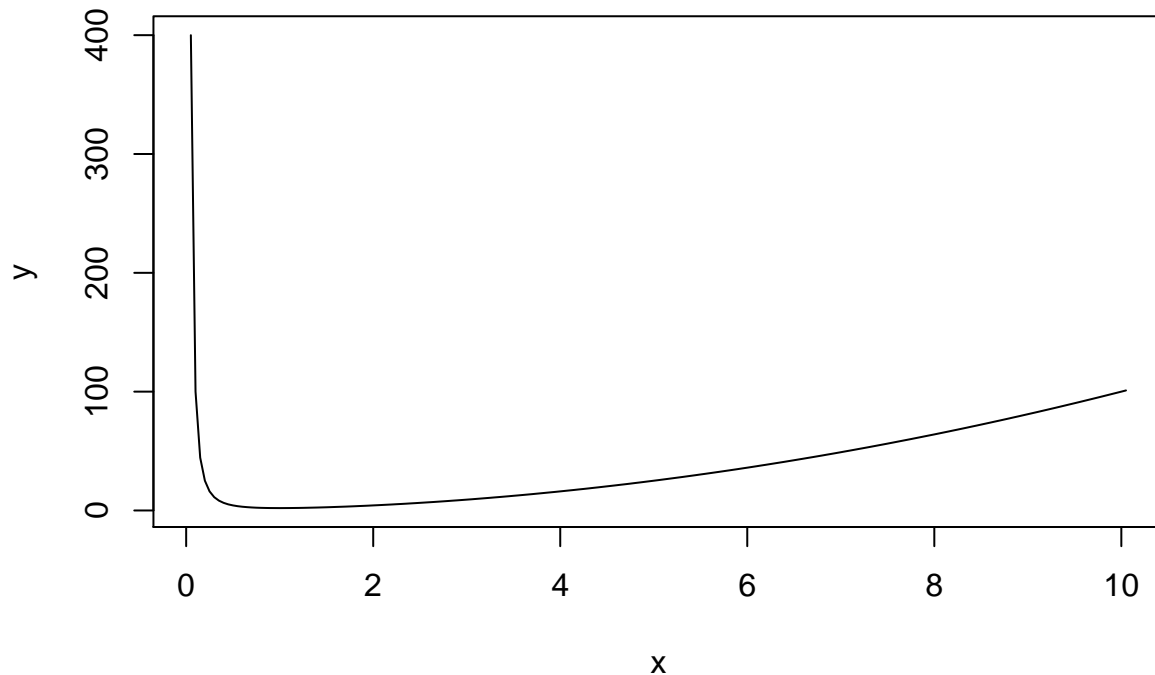
This function was developed by one of us to provide a simple but (for  $n$  equal 1 or 2) graphic example of a function with an infinity of solutions for  $n \geq 2$ . The function can be seen by graphing it to have a spike in the “middle” of a dish, much like some older candlesticks or candle holders. 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 <- optimr(x0, cstick.f, cstick.g, method="Rvmmin", control=list(trace=1))
```

```
## Unit parameter scaling
## 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::hessian(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::grad(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::hessian(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::jacobian(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::jacobian(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     70   119           92 TRUE TRUE
## snewtonm    1  1  1  1 1.399599e-28     50    88            0 TRUE TRUE
## nlm        1  1  1  1 1.004943e-16     NA   335            0 TRUE TRUE
##           xtime
## snewton    0.016
## snewtonm    0.012
## nlm        0.020

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     67   116            0 TRUE TRUE
## snewtonm    1  1  1  1 0.000000e+00     50    81            0 TRUE TRUE
## nlm        1  1  1  1 1.004942e-16     NA   335            0 TRUE TRUE
##           xtime
## snewton    9.212
## snewtonm   6.844
## nlm       51.568
```

## 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??).

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::grad(ofn)

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

##    user  system elapsed
## 0.576   0.000   0.577

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

##    user  system elapsed
## 0.044   0.000   0.043

ogr1 <- autodiffr::grad(ofn, xsize = runif(100))

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

##    user  system elapsed
## 0.032   0.000   0.030

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

##    user  system elapsed
## 0.024   0.000   0.028
```

```
ogr2 <- autodiffr::grad(ofn, xsize = runif(100), use_tape = TRUE)
```

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

```
##      user  system elapsed
```

```
## 0.348 0.000 0.349
```

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

```
##      user  system elapsed
```

```
## 0.012 0.000 0.012
```

## 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.