Optimization

malaikah bashir

08/05/2021

## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

# load required packages

library('ggplot2') library('plotly') library("shiny")

gss <- function(f, a, b, it = 10, tol = 1e-05 ){

# ' Golden-section search function

# ' @param f A function that takes one argument x that is to be optimized.

# ' The function must be strictly unimodal on the interval [a, b], if there is a unique value in [a,b] such that is the minimum of f is on [a,b]

# ' @param a The lower bound of the function, in which the optimal value is larger than. This value must also be real and scalar.

# ' @param b The upper bound of the function, in which the optimal value is smaller than. This value must also be real and scalar.

# ' @param it Number of iterations that the algorithm should run. Default is set at 10.

# ' @param tol A tolerance level of which is used to show that the function is convergingtowards the optimal solution. Set at 1e-05

# ' @export

# ' @examples

# ' > gss(f, 1, 5)

# ' Interation number 10

# ' The subset interval 1.989217 1.996894

# ' The estimated minimizer 1.993056

#calculate the golden ratio golratio <- (sqrt(5) - 1)/2

#set initial points using golden ratio c <- b - golratio \* (b - a) d <- a + golratio \* (b - a)

#evaluate the fn at the test points f\_c = f(c) f\_d = f(d) iteration = 0

min <- rep(0, length(it)) upper <- rep(0, length(it)) lower <- rep(0, length(it))

oglower <- a ogupper <- b func <- f

while (abs(b - a) > tol && iteration < it){ iteration = iteration + 1

if (f\_d > f\_c)  
 # min is then smaller than d  
 # let d be the new upper bound  
 # let c be the upper test point  
 {  
 # set d as new upper bound  
 b = d  
  
 # set new upper test point  
 d = c  
 f\_d = f\_c  
  
 #set new lower test point  
 c <- b - golratio \* (b - a)  
 f\_c = f(c)  
}  
else  
 # min is then smaller than c  
 # let c be the new lower bound  
 # let d be the lower test point  
 {  
 # set c as new lower bound  
 a = c  
 #set new lower test point  
 c = d  
 f\_c = f\_d  
 # set new upper test point  
 d <- a + golratio \* (b - a)  
 f\_d = f(d)  
 }  
  
#create classes for the upper interval,  
#lower interval and minimum of each iteration of the algorithm  
  
upper[iteration] = d  
lower[iteration] = c  
min[iteration] = (c(c+d)/2)  
  
golden = list(a = oglower , b = ogupper, func = func, c = c, d = d, upperinterval = upper, lowerinterval = lower, root\_x = min, iteration = iteration)  
class(golden) = c("golden", class(golden))

}

golden } plot.golden <- function(x, ...) {

#create a loop for the plot of f xi <- seq(xb, by = ((xa)/100)) fx <- rep(0, length(xi)) for (i in 1:length(xi)) { fx[i] = x$func(xi[i]) } xy <- setNames( data.frame(xi, fx), c("x", "y") )

#create the plot for function f # for(i in 1:length(x$minimum)){ fig <- plot\_ly(xy, x= ~x, y = ~y, mode = 'lines', type = 'scatter') fig <- fig %>% add\_trace(x = (x$root\_x[i]), y = xroot\_x[i])), type = 'scatter')

#plot( x= xi, y = fx, type = 'l', xlab = 'x', ylab = 'f(x)', main = "Golden Section Search Algorithm") +  
#points(x = (x$lowerinterval[i]), y = f(x$lowerinterval[i]), pch = '|', col = 'green' )  
#points(x = (x$upperinterval[i]), y = f(x$upperinterval[i]), pch = '|', col = 'green')  
#points(x = (x$minimum[i]), y = x$fun((x$minimum[i])), pch = 18, col = 'blue')

# readline('Press ENTER to continue...')

# }

}

gs <- gss(f,2,5)

summary.golden <- function(x,...){

result\_it <- setNames( data.frame(xupperinterval, x$root\_x), c("","Subset Interval", "Minimum" ) )

cat("Results after each iteration", '') print(result\_it) } print.golden <- function(x,...){

cat( 'Iteration number', x$iteration, '\n', 'The subset interval', c(x$c, x$d), '\n', 'The estimated minimizer', (x$c + x$d)/ 2, '') }

# example function for GSS

f <- function(x){ return((x-2)\*\*2) } gs <- gss(f, 1, 5)

brents <- function(f, a, b, iteration = 20, tol = 1e-05){

# ' Brent's Method for Optimization

# ' @param f A function that takes one argument x that is to be optimized

# ' Note that the function has to be continuous over the interval stated and must have a Lagrange interpolating polynomial of 2 or more

# ' @param a Known as the contrapoint such that f(a) and f(b) need to contain different signs to each other and the interval contains the solution.

# ' @param b The current iterate of the algorithm that is the current guess for the root of the function.

# ' @param iteration Number of iterations that the algorithm should run. Default is set at 20.

# ' @param tol A tolerance level of which is used to show that the function is convergingtowards the optimal solution. Default is set at 1e-05

# ' @export

# ' @examples

# ' > brents(g, 2, 5, 16)

# ' Number of iterations 16

# ' Root 4.472153

#calculate f(a) and f(b) f\_a = f(a) f\_b = f(b)

func <- f lower <- a current\_guess <- b

if(f\_a \* f\_b >= 0){ return("Signs of f(a) and f(b) must be opposites !") #throws exception if root isn't bracketed }

if(abs(f\_a) < abs(f\_b)){ a = current\_guess b = lower f\_a = f\_b f\_b = f\_a } c = a #c = largest of the upper and lower bounds mflag = TRUE steps = 0 f\_c = f\_a value <- rep(0, length(iteration))

while(steps < iteration && abs(b - a) > tol ){

steps = steps + 1  
  
 if(f\_a != f\_c && f\_b != f\_c){  
 l0 = (a \* f\_b \* f\_c) / ((f\_a - f\_b) \* (f\_a - f\_c))  
 l1 = (b \* f\_a \* f\_c) / ((f\_b - f\_a) \* (f\_b - f\_c))  
 l2 = (c \* f\_a \* f\_b) / ((f\_c - f\_a) \* (f\_c - f\_b))  
 s = l0 + l1 + l2 #inverse quadratic interpolation  
 }  
 else{  
 s = b - (f\_b\*((b - a)/ (f\_b - f\_a))) #secant  
 }  
 if ( (s < ((3\*a + b)/ 4) | s > b) |  
 (mflag = TRUE && (abs(s - a) >= (abs(b - c)/2))) |  
 (mflag = FALSE && (abs(s - b) >= (abs(c - d)/2))) |  
 (mflag = TRUE && (abs(b - c)< tol)) |  
 (mflag = FALSE && (abs(c - d) < tol)) ) {  
 s = (a + b)/ 2 #bisection  
 mflag = TRUE  
 }  
 else{  
 mflag = FALSE  
 }  
 f\_s = f(s)  
 d = c  
 c = b  
 if(f\_a \* f\_s < 0){  
 b = s  
 }  
 else{  
 a = s  
 }  
 if(abs(f\_a) < abs(f\_b)){  
 a = b  
 b = a  
 }  
 value[steps] = c(s)  
 bmethod = list(lower = lower, current\_guess = current\_guess, root\_x = value, func = func, final\_root = s, iterations = steps)  
 class(bmethod) = c("BrentsMethod", class(bmethod))  
}

bmethod } plot.BrentsMethod <- function(x, ...) {

#create a loop for the plot of f xi <- seq(xcurrent\_guess+3, by = ((xlower)/100)) fx <- rep(0, length(xi)) for (i in 1:length(xi)) { fx[i] = x$func(xi[i]) } xy <- setNames( data.frame(xi, fx), c("x", "y") )

#create the plot for function f fig <- plot\_ly(data =xy, x = ~x, y = ~y, mode = 'lines', type = 'scatter') #for(i in 1:length(x$root)){ fig <- fig %>% add\_trace(x = x$final\_root, y = xfinal\_root), mode = 'marker')

# points(x = (x$root[i]), y = x$func(x$root[i]), pch = 'x', col = 'red' )

# readline('Press ENTER to continue...')

# } fig

} plot(bq) plot(exampbrents)

summary.BrentsMethod <- function(x,...){

result\_it <- setNames( data.frame(x$root), c("Root after each iteration" ) ) print(result\_it) } print.BrentsMethod <- function(x,...){

cat('Number of iterations', x$iterations, '\n') cat('Root', x$final\_root, '')

}

# test fn 1

g <- function(x){ return((x\*\*2)-20) } bq <- brents(g, -2, 5, 16)

# test 2

brentex <- function(x){ return((x+3)\*((x+1)\*\*2)) } exampbrents <- brents(brentex, -4, 4/3)

newtons <- function(f, x0, it = 20, tol = 1e-07 ){

#' Newton's Method #' #' @param f Function f that takes one variable x #' @param x0 The intial guess for the root of the function #' @param it Number of iterations the algorithm should run, default set at 10 #' @param tol Tolerance level, deafault set at 1e-07 #' #' @return root of the function and the number of iterations ran #' @export #' #' @examples #' #' z <- function(x){ #' return(cos(x) - x\*\*3 ) #' } #' #' > newtons(z, 0.5) #' Number of iteration 6 #' Solution 0.865474 #'

#intial values h <- 0.0001 iteration = 0

#set up variables for s3 methods root\_x <- rep(0, length(it)) root\_y <- rep(0, length(it)) df\_iter <- rep(0, length(it)) fun <- f

while (iteration != it) {

#calculate y value  
y = f(x0)  
iteration = iteration + 1  
  
#save each root for s3 methods  
root\_x[iteration] = x0[1]  
root\_y[iteration] = x0[2]  
  
#calculate the differential for the algorithm  
df = (f(x0+h) - f(x0))/h  
#save differential of each iteration  
df\_iter[iteration] = df  
  
if(abs(c(df)) < tol ){  
 break  
 #stop algorithm if the denominator is too small  
}  
  
#newtons function  
x1 <- x0 - (y/df)  
solfound = TRUE  
  
if(abs(sum(x1 - x0)) <= tol){  
 #stop algorithm when the result is within the tolerance  
 solfound = TRUE  
 break  
}  
  
x0 = x1 #update the value of x0 to current root value  
sol = list(func = f, df = df\_iter, root\_x = root\_x, root\_y = root\_y, intial\_x = x0[1], intial\_y = x0[2], sol = x0, it = iteration)

} if(solfound == TRUE ){

} else{ print('Algorithm did not converge') } class(sol) = c('newtons', class(sol)) return(sol) } plot.newtons <- function(x, ...) {

#generates plotfor 2D functions if(is.na(x$intial\_y) == TRUE){

#create a loop for the plot of f  
xi <- seq(x$intial\_x-5, x$intial\_x+5, by = 0.01)  
fx <- rep(0, length(xi))  
for (i in 1:length(xi)) {  
 fx[i] = x$func(xi[i])  
}  
  
#create the plot for function f  
for(i in 1:length(x$x)){  
 plot(xi, fx, type = 'l', xlab = 'x', ylab = 'f(x)', main = "Newton's Method") +  
 points(x = (x$x[i]), y = x$func(x$x[i]), pch = 'x', col = 'red' )  
  
 readline('Press ENTER to continue...')  
  
}

} #generates 3d plots

else{ #access required for 3d plotting library(plotly)

#create a loop for the plot of f  
xi <- seq(x$intial\_x-1, x$intial\_x+1, by = 0.1)  
yi <- seq(x$intial\_y-1, x$intial\_y+1, by = 0.1)  
xy <- expand.grid(xi, yi)  
  
#create list of z values using function  
f\_xy <- rep(0, length(xi))  
for (i in 1:nrow(xy)) {  
 f\_xy[i] <- x$fun(xy[i,])  
}  
zi <- data.matrix(f\_xy)  
  
#merge all data into one data frame  
plotdata <- setNames(  
 data.frame(xy, zi),  
 c("x", "y", "z")  
)  
  
#create the plot for function f  
library(plotly)  
plot\_ly(plotdata, x = ~x, y = ~y, z = ~z, type="mesh3d",  
 intensity = ~z,  
 colorscale = 'Viridis')

}

} summary.newtons <- function(x,...){

result\_it <- setNames( data.frame(xroot\_x, x$root\_y), c( "Differential","x", "y" ) )

cat("Results after each iteration", '') print(result\_it)

} print.newtons <- function(x,...){

cat('Number of iterations taken', x$it, '\n') cat('Solution', x$sol, '')

}

# beale function

beale <- function(x){ return((1.5 - x[1] + x[1]\*x[2])**2 + (2.25 - x[1] + x[1]*(x[2]2))2 + (2.625 - x[1] + x[1]*(x[2]**3))\*\*2) } #beale fn nra <- newtons(beale, c(3.1,0.55))

# matyas function

matyas <- function(x){ return(0.26*(x[1]****2 + x[2]****2) - 0.48*x[1]\*x[2] ) } newtex <- newtons(matyas, c(0.02, 0.05))

gradient <- function()

sgd <- function(x, y, alpha = 0.01, it = 50){

#bind together x and y vectors to enable accurate sampling xy <- data.frame(cbind(y,x)) #set up intial theta matrix, representing the theta <- matrix(c(0,0,0,0,0) , nrow = 1) #matrix to store results from each iteration alltheta <- matrix(NA, nrow = it, ncol = 3)

set.seed(25) #xy data sample from data samp <- as.matrix( xy[sample(nrow(xy), 5, replace = TRUE), ])

#define the x and y sample matricies samp\_x <- as.matrix(samp[,2]) samp\_y <- as.matrix(samp[,1]) for(i in 1:it){

print(samp\_x)  
print(theta)  
  
  
#calculate the prediction value using sample data  
prediction <- (1/nrow(samp\_y)) \* ((theta) %\*% t(samp\_x))  
  
#update the theta value  
theta <- theta - alpha \* ((prediction) - t(samp\_y))  
  
#save the theta value  
#alltheta[i,] <- theta

} return(theta) } attach(mtcars) sgd(disp, mpg)

# ' Log-Barrier Method

# '

# ' @param f Objective function f to which the user is trying to solve for

# ' @param cons Inequality/equality contraints in the form such that Ax = b is Ax - b

# ' @param x0 Intial guess of the solution to the optimization problem

# ' @param it Number of iterations, default is set at 10

# ' @param m Highest number of dimensions in the problem, eg. x^2 + 2x would be 2

# ' @param mu multiple to increase t by, must be greater than 1

# ' @param epsilon default set at 1e-07

# '

# ' @return

# ' The solution and number of iterations ran, can be called using the print function

# ' @export

# '

# ' @examples

# '

# ' ex1 <- function(x){return((1 - x[1])\*\*2 + 100\*(x[2] - x[1]**2)**2)} #' ex2 <- function(x){return((x[1] - 1)\*\*3 - 1)} #' lbexample <-logbarrier(ex1, ex2, c(1.2, 1.2), m = 2, mu = 3) #' print(lbexample) #' #' log-barrier solution 1.00001 1.00001 #' Number of iterations 10

logbarrier <- function(f, cons, x0, it = 10, m, mu, epsilon = 1e-07){

#intial values x\_t = x0 t = m/ epsilon iteration = 0 f = f cons = cons root\_x <- rep(0, length(it)) root\_y <- rep(0, length(it)) resultf <- rep(0, length(it))

#create function of problem to be solved problem <- function(x){ return(f(x) - t*(log(-1*(cons(x))))) }

while(iteration != it){

#set up iteration counter  
iteration = iteration + 1  
  
if (m/t > epsilon){  
 print("Algorithm did not converge")  
}  
  
else{  
  
 resultf[iteration] = problem(x\_t)  
 #use newtons method to extract the root  
 newt <- newtons(problem, x\_t, it = it)  
 x <- newt$sol  
  
 #update root and t value before next iteration  
 x\_t <- c(x)  
 t <- mu \* t  
 root\_x[iteration] = x\_t[1]  
 root\_y[iteration] = x\_t[2]  
  
}  
barrier = list(func = f, constraint = cons, intial = x0,  
 x\_root = root\_x, y\_root = root\_y, sol = x\_t,  
 result\_function = resultf, it = iteration)  
class(barrier) = c('barrier', class(barrier))

} barrier } plot.barrier <- function(x, ...) {

#generates error message for 2D functions if(is.na(x$y\_root[1]) == TRUE){ return("Plot cannot be produced, for plotting please use a 3-dimentional function") }

#access required for 3d plotting library(plotly)

#create a loop for the plot of f xi <- seq(xintial[1]+3, by = 0.1) yi <- seq(xintial[2]+3, by = 0.1) xy <- expand.grid(xi, yi)

#create list of z values for main function f\_xy <- rep(0, length(xi)) for (i in 1:nrow(xy)) { f\_xy[i] <- x$func(xy[i,]) } zi <- data.matrix(f\_xy)

#merge all data into one data frame plotdata <- setNames( data.frame(xy, zi), c("x", "y", "z") )

#create z values for constraint function c\_xy <- rep(0, length(xi)) for(i in 1:nrow(xy)){ c\_xy[i] <- x$func(xy[i,]) } c\_zi <- data.matrix(c\_xy) #merge the data for constraints together consdata <- setNames( data.frame(xy, c\_zi), c("x", "y", "z") )

#create the plot for function f library(plotly) fig <- plot\_ly(plotdata, x = ~x, y = ~y, z = ~z, type="mesh3d", intensity = ~z, colorscale = 'Viridis', opacity = 0.75) #fig <- fig %>% add\_trace(type = 'contour', # data = consdata, # x = ~x, # y = ~z

# ) fig

} summary.barrier <- function(x,...){

result\_it <- setNames( data.frame(xy\_root, x$result\_function), c("x", "y", "barrier problem") ) cat("Result after each iteration", '') print(result\_it) } print.barrier <- function(x,..){

cat("log-barrier solution", x$sol, '\n') cat("Number of iterations", x$it)

}

# Rosenbrock function constrained to a disk

ex1 <- function(x){return((1 - x[1])\*\*2 + 100\*(x[2] - x[1]**2)**2)} ex2 <- function(x){return((x[1] - 1)\*\*3 - 1)} lbexample <-logbarrier(ex1, ex2, c(1.2, 1.2), m = 2, mu = 3)

# Gomez and Levy function

pt1 <- function(x){return(4\*x[1]\*\*2 - 2.1\*x[1]\*\*4 + (1/3)\*x[1]**6 + x[1]*x[2] - 4*x[2]**2 - 1 + x[2]\*\*4)} pt2 <- function(x){return(sin(4*pi*x[1]) + 2*sin(2*pi\*x[2])\*\*2 - 1.5)} #logbarrier(pt1, pt2, c(0.8, -0.7), m = 2, mu = 3)

## Including Plots

You can also embed plots, for example:

Note that the echo = FALSE parameter was added to the code chunk to prevent printing of the R code that generated the plot.