HUDM6026 Homework_05

Chenguang Pan & Seng Lei

Feb 24, 2023

Q1:

Determine the first derivative of f and encode it in a function called f_prime.

MY SOLUTION:
Based on chain rule, the first derivative of $f(x)$ is

$$f(x)' = \frac{-2x}{x^2+1} + \frac{1}{3}x^{-\frac{2}{3}}$$

. Based on this equation, I write the code below. Certainly, we can use the R-built-in function to get the derivative quickly.

```
> f_prime <- function(x) {
+   out_ <- (-2*x)/(x^2 + 1) + (1/3)*(x^(-2/3))
+   return(out_)
+ }
> # since this question requires the maximum, I use -1 times the function.
> f_prime_neg <- function(x){
+   out_ <- (-2*x)/(x^2 + 1) + (1/3)*(x^(-2/3))
+   out_ <- out_*(-1)
+   return(out_)
+ }
```

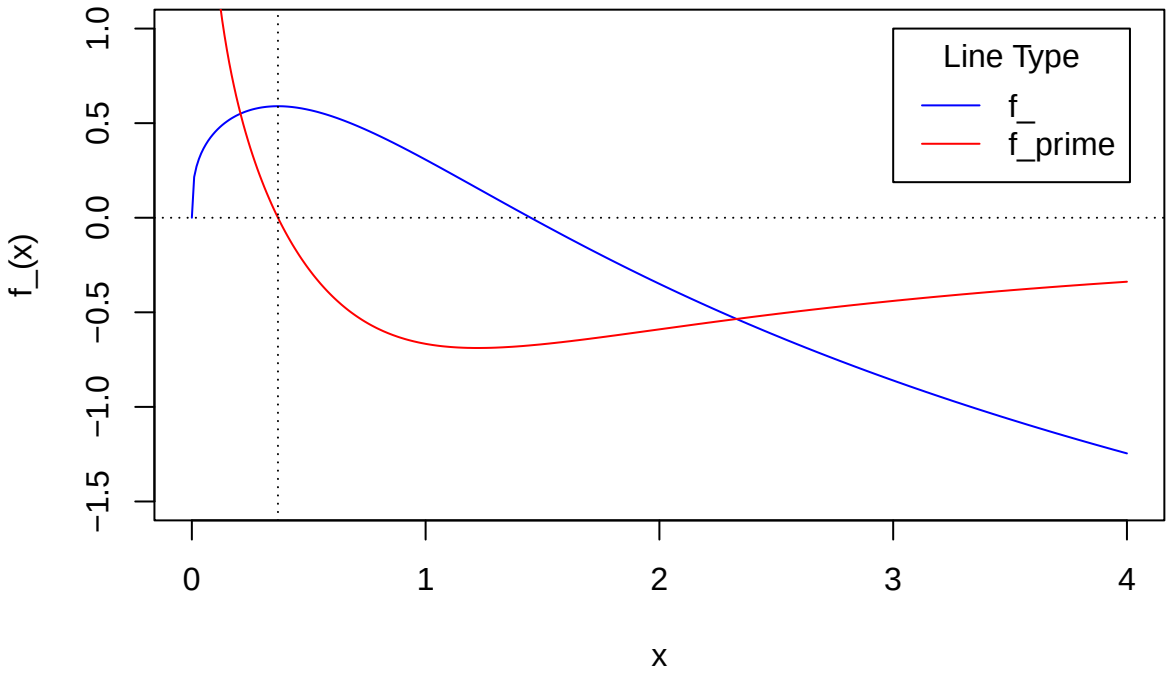
Q2:

Create a plot of f and f on [0,4] in different colors and line types and add a legend.
MY SOLUTION:

```
> # write the original function with the name of f_
> f_ <- function(x){
+   out_ <- (-1)*log(x^2 + 1) + x^(1/3)
+   return(out_)}
> # since this question requires the maximum, I use -1 times the function.
> f_neg <- function(x){
+   out_ <- (-1)*log(x^2 + 1) + x^(1/3)
+   return(out_*(-1))}
>
> # first plot the original function
> x <- seq(0,4,0.01)
> # plot the original function with blue line
> plot(x, f_(x), col="blue", type = "l", ylim = c(-1.5,1))
```

```
> # plot the first derivative with red line
> lines(x, f_prime(x), col="red", type = "l")
> # add the legend
> legend(3,1, inset = 0.1, c("f_","f_prime"), lty = 1,
+        col = c("blue","red"), title="Line Type")
> # add a horizontal line to indicate the y=0
> abline(h=0,lty=3)
> abline(v=0.3683525,lty=3)
> mtext("Figure 1. The Lines of the Given Function and it's First Derivative",
+       side = 3,
+       line = -2,
+       outer = T)
```

Figure 1. The Lines of the Given Function and it's First Derivative



Q3:

Finish the functions that I started in the R code notes for univariate optimization for the golden section search, the bisection method, and Newton's method.

MY SOLUTION:

3.1 The Golden Section Search

```
> golden <- function(f, int, precision = 1e-6)
+ {
+   rho <- (3-sqrt(5))/2 # ::: Golden ratio
+   # ::: Work out first iteration here
+   f_a <- f(int[1] + rho*(diff(int)))
+   f_b <- f(int[2] - rho*(diff(int)))
+   ### How many iterations will we need to reach the desired precision?
+   N <- ceiling(log(precision/(diff(int)))/log(1-rho))
+   for (i in 1:(N))                    # index the number of iterations
+   {
+     if (f_a < f_b)
+     {
+       int[2] <- int[2] - rho*(diff(int))
+       f_b <- f(int[2])
+     } else{
+       if (f_a >= f_b)
+       {
+         int[1] <- int[1] + rho*(diff(int))
+         f_a <- f(int[1])
+       } }
+   }
+   int
+   print(paste0("Iteration for ",N," times;",
+                " The maximum value is at", round(int[1],6)))
+ }
```

3.2 The Bisection Method

More information about this method can be found on *Page 116* of Chong and Zak (2013).

```
> bisection <- function(f_prime, int, precision = 1e-7)
+ {
+   # ::: f_prime is the function for the first derivative
+   # ::: of f, int is an interval such as c(0,1) which
+   # ::: denotes the domain
+
+   N <- ceiling(log(precision/(diff(int)))/log(.5))
+   # find the midpoint of the initial uncertainty range
+   midpoint <- (int[1]+int[2]) /2
+   # evaluate the f_prime on the midpoint
+   f_prime_a <- f_prime(midpoint)
+   i <- 1
+   for (i in 1:N)
+   {
+     i <- i + 1
+     if(f_prime_a < 0)
+     {
+       # if the f_prime on the midpoint is less than 0,
+       # the minimizer must be on the right side of midpoint.
+       int[1] <- midpoint
+       # update the uncertainty range
+       midpoint <- (int[1]+int[2]) /2
```

```
+     } else
+       if(f_prime_a > 0)
+       {
+         # if the f_prime on the midpoint is less than 0,
+         # the minimizer must be on the left side of midpoint.
+         int[2] <- midpoint
+         midpoint <- (int[1]+int[2]) /2
+       } else
+         if(f_prime_a == 0)
+         {
+           break
+         }
+     # ::: FILL IN CODE HERE (UPDATE)
+     f_prime_a <- f_prime(midpoint)
+   }
+   int
+   print(paste0("Iteration for ",N," times;",
+                " The maximum value is at ", round(int[1],6)))
+ }
```

3.3 The Newton's Method

More information about this method can be found on *Page 116* of Chong and Zak (2013).

```
> newton <- function(f_prime, f_dbl, precision = 1e-6, start)
+ {
+   x_old <- start
+   x_new <- x_old - f_prime(x_old)/f_dbl(x_old)
+
+   i <- 1
+   print(paste0("Iteration ", i, "; Estimate = ", x_new) )
+   while (abs(x_new-x_old) > precision){
+     x_old <- x_new
+     x_new <- x_old - f_prime(x_old)/f_dbl(x_old)
+     # ::: redefine variables and calculate new estimate
+     # ::: keep track of iteration history
+     print(paste0("Iteration ", i+1, "; Estimate = ", x_new) )
+     i <- i + 1
+   }
+   x_new
+ }
```

Q4:

Apply each of the three functions to this example to discover the minimum. Keep track of and report the number of iterations required for each method. Report the coordinates of the minimum discovered by each of the three functions as well as the number of iterations required

MY SOLUTION:

4.1 Apply the Golden Section Search

```
> golden(f_neg, c(0,4))
[1] "Iteration for 32 times; The maximum value is at0.368352"
```

4.2 Apply the Bisection Method

```
> bisection(f_prime_neg, c(0,4))
[1] "Iteration for 26 times; The maximum value is at 0.368352"
```

4.3 Apply the Newton's Method

First, I need to get the second derivative of the original function. Based on the Quotient Rule and Chain Rule, one can easily have

$$f(x)'' = -6x^2 - 2 - \frac{2}{9}x^{-\frac{5}{3}}$$

```
> # write the second derivative function
> f_dbl <- function(x){
+   out_ <- -6*x^2-2-(2/9)*x^(-5/3)
+   return(out_)
+ }
> # since this question requires the maximum, I use -1 times the function.
> f_dbl_neg <- function(x){
+   out_ <- -6*x^2-2-(2/9)*x^(-5/3)
+   return(out_*(-1))
+ }
```

Plug the first and the second derivatives into the Newton's Method function.

```
> newton(f_prime_neg, f_dbl_neg,start = 1)
[1] "Iteration 1; Estimate = 0.918918918918919"
[1] "Iteration 2; Estimate = 0.830999851550479"
[1] "Iteration 3; Estimate = 0.736988742573808"
[1] "Iteration 4; Estimate = 0.639870150729756"
[1] "Iteration 5; Estimate = 0.546642635261104"
[1] "Iteration 6; Estimate = 0.468667930513546"
[1] "Iteration 7; Estimate = 0.416016821629135"
[1] "Iteration 8; Estimate = 0.388211559563474"
[1] "Iteration 9; Estimate = 0.376060038291039"
[1] "Iteration 10; Estimate = 0.371254711645403"
[1] "Iteration 11; Estimate = 0.369432581306114"
[1] "Iteration 12; Estimate = 0.368752709077139"
[1] "Iteration 13; Estimate = 0.368500562445562"
[1] "Iteration 14; Estimate = 0.368407257199536"
[1] "Iteration 15; Estimate = 0.368372758807196"
[1] "Iteration 16; Estimate = 0.36836000738855"
[1] "Iteration 17; Estimate = 0.368355294697857"
[1] "Iteration 18; Estimate = 0.36835355304667"
[1] "Iteration 19; Estimate = 0.368352909401224"
[1] 0.3683529
```

All three methods have the identical results, that is the maximum point of the given function is at $x = .368352$, as shown in `Figure.1`. As for the iteration times at the x interval $[0, 4]$, the Golden Section Search iterated 32 times; the Bisection Method iterated 26 times. However, Newton's method iteration time largely depends on the start numbers. That is, the more guessing number close to the target, the less times needed.

Q5:

Did the methods perform as you expected in terms of the number of iterations? Why or why not?

MY SOLUTION:

Yes. The bisection method has larger "step" than golden search method, therefore, the search iteration should be more efficient for bisection method. However, Newton's method iteration time largely depends on the start numbers. That is, the more guessing number close to the target, the less times needed.

Q6:

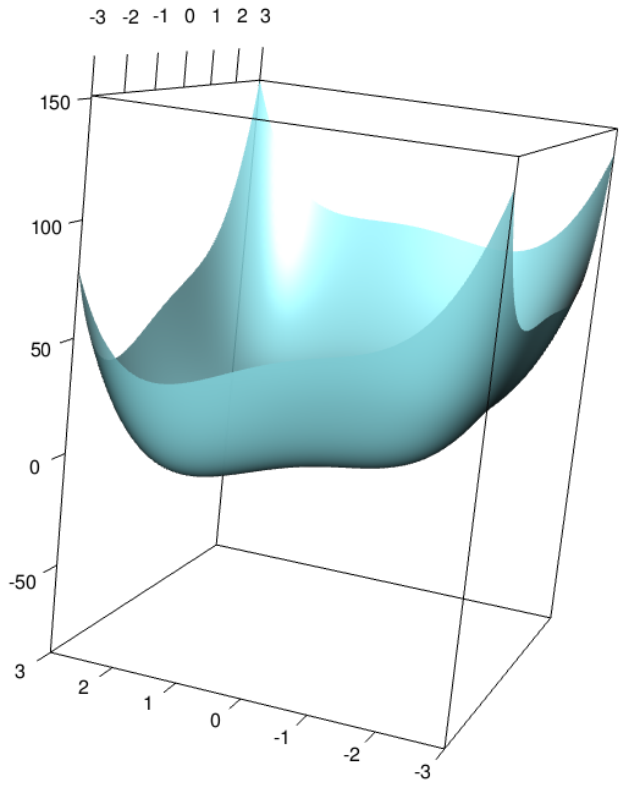
Use function persp3d() in package rgl to plot function f on [-3,3]. . . .

MY SOLUTION:
I write the function first.

```
> # write the function
> f_bi <- function(x_1, x_2){
+   out_ <- x_1^4 + x_2^4 - 2*x_1^2 + 2*x_1*x_2 - 3*x_2^2 + 6*x_1 - 4*x_2 + 10
+   return(out_)
+ }
```

Then, plot the 3-d surface of function.

```
> library(rgl)
> # Define grid for plotting
> x = seq(-3, 3, by = .05)
> y = seq(-3, 3, by = .05)
> xy <- expand.grid(x,y)
>
> # Calculate height of function
> # xy[,1] is vector of 'x' values; xy[,2] is vector of 'y' values
> z <- mapply(FUN = f_bi, xy[,1], xy[,2])
>
> ### Plot the surface over the domain
> plot3d(xy[,1], xy[,2], z, type = "n", radius = 1.5,
+        col = "cadetblue1", zlim = c(-90, 90), xlab = "",
+        ylab = "", zlab = "")
> surface3d(x, y, z = matrix(z,length(x)), col = "cadetblue1",
+           zlim = c(-90, 90), alpha = .9)
> knitr::include_graphics("plot-3d.png")
```



From the 3D plot, I guess the minimum point is around (-2.1, 1.8, -11).

Q7:

Find the partial derivatives of f with respect to x1 and x2 and write a function called grad_f() that gives the gradient of f.

MY SOLUTION:

```
> # write the original function
> f_origin <- function(x_1, x_2){
+   out_ <- x_1^4+x_2^4-2*x_1^2+2*x_1*x_2-3*x_2^2+6*x_1-4*x_2+10
+   return(out_)
+ }
>
> # write the two partial derivatives
> dfdx_1 <- function(x_1, x_2){
+   return(4*x_1^3-4*x_1 + 2*x_2+6)
+ }
> dfdx_2 <- function(x_1, x_2){
+   return(4*x_2^3+2*x_1-6*x_2-4)
+ }
>
> # write the gradient descent function
> grad_f <- function(x_1, x_2){c(dfdx_1(x_1,x_2), dfdx_2(x_1,x_2))}
```

Q8:

Write a function to implement gradient ascent with arguments for the start point, the number of iterations, the step size alpha...

MY SOLUTION:
Here, we need to discuss about the necessity of normalized gradient. Intuitively, if each time we normalize the gradient, the length of "step size" for updating is fixed. That is, the value of $Learning\_Rate * Gradient's\_Unit\_Vector$. Apparently, it is a fixed rather than dynamic value to update the x_1 and x_2 coordination. Therefore, one can easily see the final steps will always oscillate around the target point and the stop criterion might never be reached! I choose to skip the `normalize` step, and use the $Learning\_Rate*Gradient$ directly as a dynamic updating value to avoid the oscillation. That is, the closer to the target point, the smaller the step size.

```
> gradient_d <- function(f_original, # the original function
+                        grad_f, # the gradient function
+                        start_point, # the start point, a vector
+                        max_iter=100, # the maximum number of iteration
+                        alpha=0.03, # learning rate/ step size
+                        epsilon=1e-5 # stopping criterion
+ ){
+   # initial settings
+   p_old <- start_point;i <- 1;check <- 1
+   while (check > epsilon) {
+     # print the iteration information at each 10 rounds
+     # if (i ==1 | i%%5 == 0){
+     #  print(paste0("Iter ", i, "; f(x_1, x_2)= ", f_original(p_old[1], p_old[2])))}
+     print(paste0("Iter ", i, "; f(x_1, x_2)= ", f_original(p_old[1], p_old[2])))
+     # Stop condition and warning
+     if (i > max_iter) {
+       print("Exceed maximum number of iterations")
+       break
+       }
+     if (abs(p_old[1]) > 3 |abs(p_old[2] >3)) {
+       print("Exceed the Given Range")
+       break
+       }
+     # load the gradient
+     gradient_ <- grad_f(p_old[1], p_old[2])
+     # As discussed above, I choose to not use the normalized gradient here
+     # gradient_norm <- gradient_/sqrt(gradient_%*%gradient_)
+     # update the point coordination with gradient * learning rate
+     p_new <- p_old - alpha*gradient_
+     # check the updating rate of p_new, if less than epsilong, stop
+     check <- sqrt((p_new-p_old)%*%(p_new-p_old))/ sqrt(p_old %*% p_old)
+     # redefine the old point to send for next updating
+     p_old <- p_new
+     i <- i + 1
+   }
+   print(paste0("The minimum point is around ",
+                round(p_old[1],4)," ",
+                round(p_old[2],4)," ",
+                round(f_original(p_old[1],p_old[2]),4)))
+ }
```

Then, try this gradient descent method on the given function.

```
> gradient_d(f_origin, grad_f, c(1,2))
[1] "Iter 1; f(x_1, x_2)= 15"
[1] "Iter 2; f(x_1, x_2)= 7.81301856"
[1] "Iter 3; f(x_1, x_2)= 6.13673240489959"
[1] "Iter 4; f(x_1, x_2)= 4.45179110852303"
[1] "Iter 5; f(x_1, x_2)= 2.46623606070743"
[1] "Iter 6; f(x_1, x_2)= -0.020016603013886"
[1] "Iter 7; f(x_1, x_2)= -3.06809730294683"
[1] "Iter 8; f(x_1, x_2)= -6.35515915651194"
[1] "Iter 9; f(x_1, x_2)= -9.0349641871389"
[1] "Iter 10; f(x_1, x_2)= -10.4007619836912"
[1] "Iter 11; f(x_1, x_2)= -10.7691610996711"
[1] "Iter 12; f(x_1, x_2)= -10.8215165666176"
[1] "Iter 13; f(x_1, x_2)= -10.8265520327193"
[1] "Iter 14; f(x_1, x_2)= -10.826987724242"
[1] "Iter 15; f(x_1, x_2)= -10.8270254054114"
[1] "Iter 16; f(x_1, x_2)= -10.8270286959674"
[1] "Iter 17; f(x_1, x_2)= -10.8270289848232"
[1] "Iter 18; f(x_1, x_2)= -10.8270290102356"
[1] "The minimum point is around -1.5711 1.6143 -10.827"
```

For text space concern, I choose to print the iteration information at each 10 rounds.The final result shows that the minimum point is around (-1.5711, 1.6143, -10.827), which is close to the observed possible minimum point (-2.1, 1.8, -11) (in my response to Question 6).

Q9:

Discuss what changes would need to be made to the gradient ascent function to im- plement steepest ascent. You do not need to actually implement steepest ascent here but you can do so for bonus points.

MY SOLUTION:
If one wants to make the steepest descent, they can adjust the learning rate(step size). But one caveat is, a big step size will cause the point oscillating around the minimum point and possibly never reaching the target. So, a dynamic learning rate is necessary. That is, the learning rate can adaptively change itself from big size at the beginning to small when closing to the target.