HUDM 6026 HW 05

Getong Zhong (gz2338) & Ruoqiao Li (rl3288)

2023-02-26

1

```r
f_prime <- function(x) -2*x/(x^2+1) + (1/3)*x^(-2/3)
```

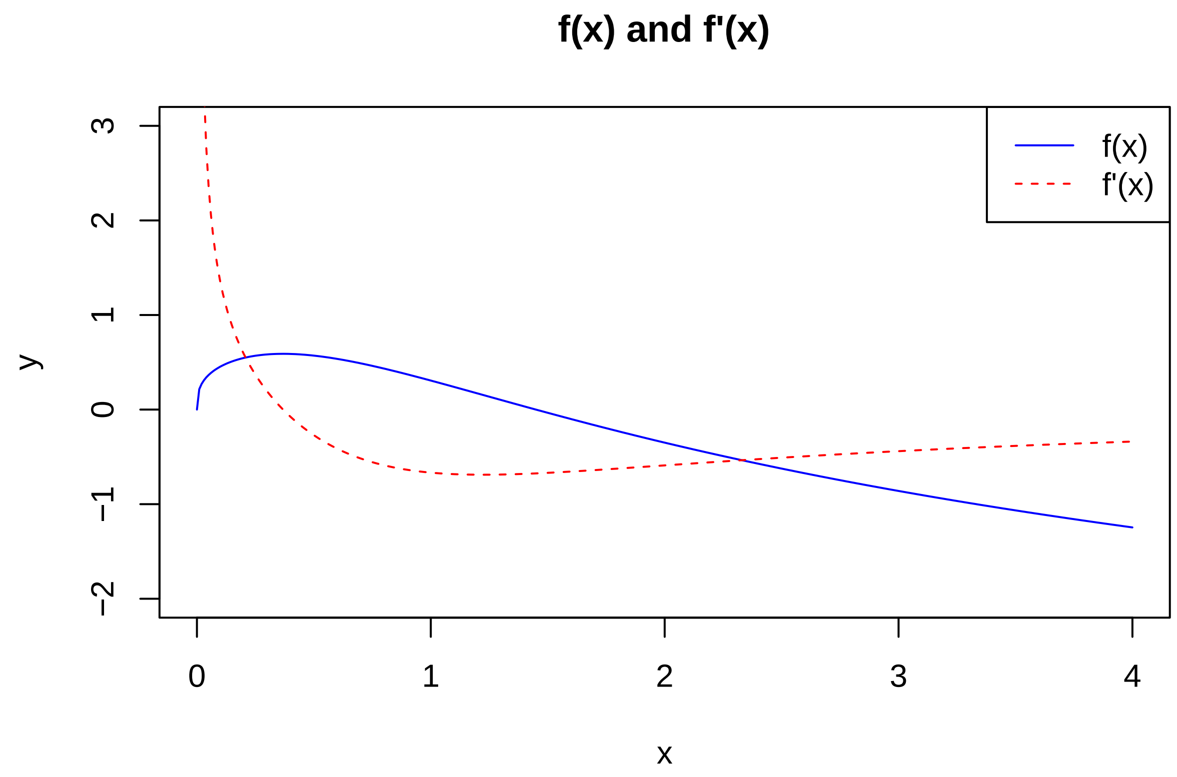
2

```r
f <- function(x) -log(x^2 + 1) + x^(1/3)

x <- seq(0, 4, 0.01)

plot(x, f(x), type = "l", col = "blue", lty = 1, ylim = c(-2, 3),
     xlab = "x", ylab = "y", main = "f(x) and f'(x)")

lines(x, f_prime(x), col = "red", lty = 2)
legend("topright", legend = c("f(x)", "f'(x)"), col = c("blue", "red"),
       lty = c(1, 2))
```



3

golden section search

```
golden <- function(f, int, precision = 1e-6)
{
  # ::: This function implements the golden section search for a
  # ::: *minimum* for the function 'f' on the range [int]
  # ::: with precision no greater than 'precision'.
  # ::: Note: 'int' is an interval such as c(2,3).
  # ::: If you want to *maximize*, multiply your function by -1.

  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[1] + rho * (int[2] - int[1])
      f_b <- f_a
      f_a <- f(int[1] + rho * (diff(int)))


    } else{
      if (f_a >= f_b)
      {
        int[1] <- int[1] + rho * (int[2] - int[1])
        f_a <- f_b
        f_b <- f(int[2] - rho * (diff(int)))
      } }
    print(paste0("Iteration ", i+1, "; Estimate = ", (f_a + f_b)/2) )
  }
  cat("Minimum found at x = ", (f_a + f_b)/2, "; Iterations: ", i, "\n")
  (f_a + f_b)/2
}
```

bisection method

```
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))
  f_prime_a <- f_prime(int[1] + diff(int)/2)
  for (i in 1:N)
  {
    if(f_prime_a < 0)
    {
      int[1] <- int[1] + diff(int)/2
      f_prime_a <- f_prime(int[1] + diff(int)/2)
    } else if(f_prime_a > 0)
      {
        int[2] <- int[2] - diff(int)/2
        f_prime_a <- f_prime(int[1] - diff(int)/2)
      } else
        {
          break
        }
    if(diff(int) < precision)
    {
      break
    }
    print(paste0("Iteration ", i+1, "; Estimate = ", f_prime_a ))
  }
  cat("Minimum found at x = ", int[1] + diff(int)/2, "; Iterations: ", i, "\n")
  int[1] + diff(int)/2
}
```

newton's method

```r
newton <- function(f_prime, f_dbl, precision = 1e-6, start)
{
  # ::: f_prime is first derivative function
  # ::: f_dbl is second derivitive function
  # ::: start is starting 'guess'

  x_old <- start
  x_new <- x_old - f_prime(x_old)/f_dbl(x_old)

  i <- 1 # ::: use 'i' to print iteration number
  print(paste0("Iteration ", i, "; Estimate = ", x_new) )
  while (abs(f_prime(x_new)) > 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
  }
  cat("Minimum found at x = ", x_new, "; Iterations: ", i, "\n")
}
```

4

Newton's need

```r
golden(f, c(0,4))
```

```
## [1] "Iteration 2; Estimate = -0.747104841237446"
## [1] "Iteration 3; Estimate = -0.958961570967187"
## [1] "Iteration 4; Estimate = -1.07573789336833"
## [1] "Iteration 5; Estimate = -1.14325431461137"
## [1] "Iteration 6; Estimate = -1.18336000194123"
## [1] "Iteration 7; Estimate = -1.20756092822106"
## [1] "Iteration 8; Estimate = -1.22230169147233"
## [1] "Iteration 9; Estimate = -1.23133109941826"
## [1] "Iteration 10; Estimate = -1.23688107948934"
## [1] "Iteration 11; Estimate = -1.24029959727625"
## [1] "Iteration 12; Estimate = -1.24240796397424"
## [1] "Iteration 13; Estimate = -1.24370933313633"
## [1] "Iteration 14; Estimate = -1.24451298562302"
## [1] "Iteration 15; Estimate = -1.24500942680324"
## [1] "Iteration 16; Estimate = -1.24531615143223"
## [1] "Iteration 17; Estimate = -1.24550568221141"
## [1] "Iteration 18; Estimate = -1.24562280513144"
## [1] "Iteration 19; Estimate = -1.24569518590463"
## [1] "Iteration 20; Estimate = -1.24573991770717"
## [1] "Iteration 21; Estimate = -1.24576756272702"
## [1] "Iteration 22; Estimate = -1.24578464800072"
## [1] "Iteration 23; Estimate = -1.24579520717051"
## [1] "Iteration 24; Estimate = -1.24580173305429"
## [1] "Iteration 25; Estimate = -1.24580576625621"
## [1] "Iteration 26; Estimate = -1.24580825890594"
## [1] "Iteration 27; Estimate = -1.24580979944586"
## [1] "Iteration 28; Estimate = -1.245810751551"
## [1] "Iteration 29; Estimate = -1.24581133998399"
## [1] "Iteration 30; Estimate = -1.24581170365545"
## [1] "Iteration 31; Estimate = -1.24581192841672"
## [1] "Iteration 32; Estimate = -1.24581206732681"
## [1] "Iteration 33; Estimate = -1.24581215317795"
## Minimum found at x =  -1.245812 ; Iterations:  32
```

```
## [1] -1.245812
```

```
bisection(f_prime, c(0,4))
```

```
## [1] "Iteration 2; Estimate = -0.439750047743621"
## [1] "Iteration 3; Estimate = -0.383702411941231"
## [1] "Iteration 4; Estimate = -0.359826088716481"
## [1] "Iteration 5; Estimate = -0.348789292684975"
## [1] "Iteration 6; Estimate = -0.343480195109242"
## [1] "Iteration 7; Estimate = -0.34087605628152"
## [1] "Iteration 8; Estimate = -0.339586356041345"
## [1] "Iteration 9; Estimate = -0.338944569438227"
## [1] "Iteration 10; Estimate = -0.338624438443296"
## [1] "Iteration 11; Estimate = -0.338464563077442"
## [1] "Iteration 12; Estimate = -0.338384672871884"
## [1] "Iteration 13; Estimate = -0.338344739631511"
## [1] "Iteration 14; Estimate = -0.338324775976059"
## [1] "Iteration 15; Estimate = -0.338314794889409"
## [1] "Iteration 16; Estimate = -0.338309804531339"
## [1] "Iteration 17; Estimate = -0.338307309398616"
## [1] "Iteration 18; Estimate = -0.338306061843833"
## [1] "Iteration 19; Estimate = -0.338305438069335"
## [1] "Iteration 20; Estimate = -0.33830512618281"
## [1] "Iteration 21; Estimate = -0.338304970239729"
## [1] "Iteration 22; Estimate = -0.338304892268233"
## [1] "Iteration 23; Estimate = -0.338304853282497"
## [1] "Iteration 24; Estimate = -0.338304833789631"
## [1] "Iteration 25; Estimate = -0.338304824043199"
## [1] "Iteration 26; Estimate = -0.338304819169983"
## Minimum found at x =  4 ; Iterations:  26
```

```
## [1] 4
```

```
f_dbl <- function(x) (2*x^3 - 4*x)/(x^2 + 1)^2 - (2/9)*x^(-5/3)
newton(f_prime, f_dbl, precision = 1e-6, 1)
```

```
## [1] "Iteration 1; Estimate = 0.0769230769230768"
## [1] "Iteration 2; Estimate = 0.180761051960308"
## [1] "Iteration 3; Estimate = 0.334244584368924"
## [1] "Iteration 4; Estimate = 0.372045684859803"
## [1] "Iteration 5; Estimate = 0.367894522302643"
## [1] "Iteration 6; Estimate = 0.368408805584456"
## [1] "Iteration 7; Estimate = 0.368345609651913"
## [1] "Iteration 8; Estimate = 0.368353383538843"
## [1] "Iteration 9; Estimate = 0.368352427378218"
## Minimum found at x =  0.3683524 ; Iterations:  9
```

5

We think the performance of the method matched our expectations. From the above results we observed that the Newton's method converged the most quickly with only 9 iterations required. The Golden section method required 33 iterations and the bisection method required 26 iterations. Such results matched the characteristic of the those three selection method such that the Newton's method is effective, while the bisection method is more reliable but slower.

6

```r
library(rgl)
```

```
## Warning: package 'rgl' was built under R version 4.2.2
```

```r
f <- function(x1, x2) {
  x1^4 + x2^4 - 2*x1^2 + 2*x1*x2 - 3*x2^2 + 6*x1 - 4*x2 + 10
}


x1 <- seq(-3, 3, length = 50)
x2 <- seq(-3, 3, length = 50)
xy <- expand.grid(x1,x2)


z <- mapply(FUN = f, xy[,1], xy[,2])
z[z < -90] <- -90

plot3d(xy[,1], xy[,2], z, type = "n", radius = 1.5,
       col = "blue", zlim = c(-90, 90), xlab = "",
       ylab = "", zlab = "")
surface3d(x1, x2, z = matrix(z,length(x1)), col = "blue",
          zlim = c(-90, 90), alpha = .9)
```

7

```
df_dx1 <- function(x1, x2) {
  4*x1^3 - 4*x1 + 2*x2 + 6
}

df_dx2 <- function(x1, x2) {
  4*x2^3 + 2*x1 - 6*x2 - 4
}
gradf <- function(x1, x2) {
  c(df_dx1(x1, x2), df_dx2(x1, x2))
}
```

8

```
gradient <- function(f,                 # original function
                     gradf,             # gradient function
                     strtpt,            # starting point
                     maxiter = 1000,     # maximum number iterations
                     alpha = .05,        # fixed step size
                     minimize = TRUE, # set to FALSE to maximize
                     epsilon = 1e-5,  # stopping criterion
                     iterhist = TRUE) # print iteration history
{
  p_old <- strtpt; error <- 1; i <- 1
  while(error > epsilon)
  {
    if(iterhist == TRUE) print(paste0("Iter ", i, "; f(x) = ", f(p_old)))
    if(i > maxiter) stop("Exceeded maximum number of iterations")
    p_new <- gradf(p_old)
    p_new <- p_new/sqrt(p_new%*%p_new) # Normalize the gradient vector
    ### Subtract for minimization; add for maximization
    ifelse(minimize, p_new <- p_old - alpha*p_new, p_new <- p_old + alpha*p_new)
    ### Calculate stopping criterion
    error <- sqrt((p_new - p_old) %*% (p_new - p_old)) / sqrt(p_old %*% p_old)
    ### Redefine the old point to the new point for the next iteration
    p_old <- p_new
    i <- i + 1
  }
  return(p_new)
}

# Define the function
f <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  return(x1^4 + x2^4 - 2*x1^2 + 2*x1*x2 - 3*x2^2 + 6*x1 - 4*x2 + 10)
}

# Define the gradient function
grad_f <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  return(c(4*x1^3 - 4*x1 + 2*x2 + 6, 4*x2^3 + 2*x1 - 6*x2 - 4))
}

# Find the minimum
gradient(f = f, gradf = grad_f, strtpt = c(0,0), maxiter = 1000, minimize = TRUE, alpha = .05, epsilon =
```

```
## [1] "Iter 1; f(x) = 10"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 2; f(x) = 9.63137153665478"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 3; f(x) = 9.2465681949094"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 4; f(x) = 8.84562752538175"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 5; f(x) = 8.42867467150722"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 6; f(x) = 7.99591932910845"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 7; f(x) = 7.54765311236484"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 8; f(x) = 7.08424724621448"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 9; f(x) = 6.60615052443454"
## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 10; f(x) = 6.11388748833881"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 11; f(x) = 5.60805679374721"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 12; f(x) = 5.08932974414652"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 13; f(x) = 4.55844897623618"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 14; f(x) = 4.01622729071678"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 15; f(x) = 3.46354662653047"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 16; f(x) = 2.90135718103029"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 17; f(x) = 2.33067668190849"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 18; f(x) = 1.75258981927073"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 19; f(x) = 1.16824784808642"
## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 20; f(x) = 0.578868372426806"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 21; f(x) = -0.01426467654818"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 22; f(x) = -0.609800856959247"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 23; f(x) = -1.20632281571446"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 24; f(x) = -1.80234566121691"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 25; f(x) = -2.39631610122355"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 26; f(x) = -2.98661134028774"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 27; f(x) = -3.57153773428368"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 28; f(x) = -4.1493292031363"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 29; f(x) = -4.71814540703815"
## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 30; f(x) = -5.27606969609035"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 31; f(x) = -5.82110684842533"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 32; f(x) = -6.35118061743525"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 33; f(x) = -6.86413111474594"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 34; f(x) = -7.35771206210671"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 35; f(x) = -7.82958795256632"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 36; f(x) = -8.27733116950499"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 37; f(x) = -8.69841912191077"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 38; f(x) = -9.09023146687412"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is dep
##   Use c() or as.vector() instead.

## [1] "Iter 39; f(x) = -9.45004750776352"
## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 40; f(x) = -9.77504388304216"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 41; f(x) = -10.0622927045419"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 42; f(x) = -10.3087603836516"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 43; f(x) = -10.5113075448216"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 44; f(x) = -10.6666907998027"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 45; f(x) = -10.7715682266041"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 46; f(x) = -10.8225148299257"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] "Iter 47; f(x) = -10.8161032201851"

## Warning in p_new/sqrt(p_new %*% p_new): Recycling array of length 1 in vector-array arithmetic is de
##   Use c() or as.vector() instead.

## [1] -1.560478  1.597761
```

9

To implement steepest ascent, we need to calculate the gradient of the function at the current point and normalize the gradient vector to obtain a unit vector pointing in the direction of steepest ascent, after remain the similar calculation, we need to add the current point with the step size and in the direction of the steepest ascent. In general, after implement steepest ascent to the function, it can converge more quickly.