STAT6181
Computational Statistics I
Lecture 2

Optimization

Overview

- R review
- Finding the roots of an equation
- Optimizing functions

Introduction to R

R free software:

- We can run simulations
- We code ‘new’ statistical methods
- R is not particularly good at heavy computational problems
- R’s memory management is not the best in the world.

R – the basics (it's a calculator)

```
> 1+ 1
[1] 2
>
> 3*3
[1] 9
>
> 4^3
[1] 64
>
> 100/4
[1] 25
>
>
> 2*sin(60)          # R uses the default Radians!
[1] -0.6096212
>
> log(100)           #To the base e
[1] 4.60517
>
> log10(100)         #To the base 10
[1] 2
>
```

R- creation of columns

```
>x = c(1,2,3, 4, 5, 6)
>y = c(7,8,9,10,11,12)
>z = c(1,2,3)
```

```
> x + y
[1]  8 10 12 14 16 18
> x+z
[1] 2 4 6 5 7 9
> x*y
[1]  7 16 27 40 55 72
> sum(x*y)
[1] 217
> x*z
[1]  1  4  9  4 10 18
> y^x
[1]       7      64     729   10000  161051 2985984
> y-z
[1] 6 6 6 9 9 9
```

R- Creation of Matrices and columns

```
> a = matrix ( c( 1,2,3,4), 2,2)
> a
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> b = matrix ( c(1,2,3,4,5,6), 2,3)
> b
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> c = c( 1,2)
> c
[1] 1 2
> d = c(1,2,3)
> d
[1] 1 2 3
>
```

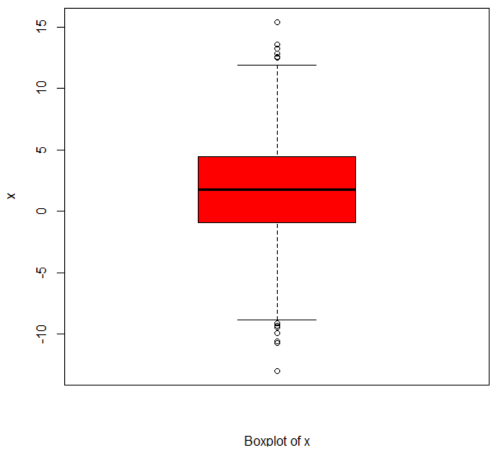
R- Matrix mathematical manipulations

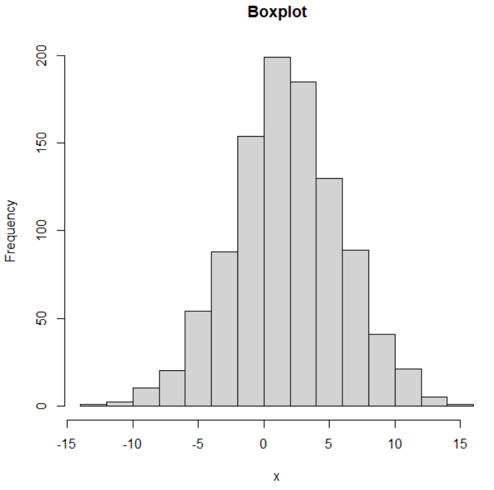
```
> a*a
     [,1]  [,2]
[1,]    1     9
[2,]    4    16
> a*b
Error in a * b : non-conformable arrays
> a%*%a
      [,1]  [,2]
[1,]    7    15
[2,]   10    22
> a%*%b
     [,1]  [,2]  [,3]
[1,]    7   15    23
[2,]   10   22    34
> a%*%t(b)
Error in a %*% t(b) : non-conformable arguments
> a%*%c
     [,1]
[1,]    7
[2,]   10
```

Plotting data – univariate data

- There are many packages that exist that can assist with graphical plots (eg ggplot etc.)
- I will go through the basics:

Univariate continuous data:






```
> x = rnorm(1000, 2,4)
> boxplot(x, col='red', xlab = 'Boxplot of x',ylab = 'x’)
> hist(x, main = "Boxplot", xlab = 'x', ylab = 'Frequency')
```

Lets just look at the distributions a bit and some R functions

These are some of the common function associated with distributions found in R.

rnorm

pnorm

dnorm

qnorm

These functions enable you to compute probabilities find generate random data from almost **any** of our ‘common’ distributions.

Some other important R functions

```
> # This is the same as computing the pdf of the normal with x = 0, mu = 0 and
> # sigma = 0. The dnorm function takes three main arguments. In this example, we are
> #calculating the “density” at x=0 from a Normal( 2, 16) distribution.
>
> dnorm(0, mean = 2, sd = 4)
[1] 0.08801633


> #The function pnorm returns the integral from −∞ to q of the pdf of the normal
> #distribution where q is a Z-score. Try to guess the value of pnorm(0,0,1).
> #pnorm has the same default mean and sd arguments as dnorm).
> # pnorm gives P(X<=x)
>
> pnorm(2, mean = 2, sd = 4)
[1] 0.5
>
> pnorm(3, mean = 2, sd = 4)
[1] 0.5987063
>
```

Some more functions - qnorm

> #The **qnorm** function is simply the inverse of the cdf. So it like the inverse of pnorm.

> #You can use qnorm to determine the answer to the question: What is the

> #qnorm answers the question - what is the Z-score of the pth quantile of the normal distribution?

>

> # What is the Z-score of the 50th quantile of the normal distribution?

> qnorm(.5)

[1] 0

>

> # What is the Z-score of the 96th quantile of the normal distribution?

> qnorm(.96)

[1] 1.750686

>

> # What is the Z-score of the 46th quantile of the normal distribution with mean 2 and variance 16?

> qnorm(.46, 2, 4)

[1] 1.598265

>

rnorm revisited

```
> #rnorm revisited
> #If you want to generate a vector of normally distributed random numbers,  > #rnorm is the function you should use. The first argument n is the number of numbers you want to generate, followed by the mean and sd arguments.
> rnorm(5, 2, 4)
[1] 5.114324 6.290950 3.102321 2.846167 4.912762
>
> rnorm(5, 2, 4)
[1] -2.3878274 -1.3153996  2.4711667  1.5269966 -0.4817649
>
> #So notice that I ran these twice and I got 5 different numbers. I need set.seed if I want the same random numbers.
> # For example,
>
> set.seed(10)
> rnorm(5, 3, 4)
[1]  3.0749847  2.2629898 -2.4853222  0.6033291  4.1781805
>
> set.seed(15)
> rnorm(5, 3, 4)
[1]  4.035291 10.324483  1.641526  6.588793  4.952065
>
> #Try set.seed = 10 now and see what happens
> set.seed(10)
> rnorm(5, 3, 4)
[1]  3.0749847  2.2629898 -2.4853222  0.6033291  4.1781805
```

Functions in R

- A function is a “set” of code that takes in objects/values/etc and returns a ‘set’ of outputs.
- The inputs can be objects in R.
- The outputs can be objects in R.

A simple R function

```
sumtwocolumns <- function( x, y)
{
z = x+y
return(z)
}
```

Lets go through line by line!!!

Now, how do you run the function

- **Step 1**: You need to highlight the code for the function and run it
- **Step 2**: You have the objects/inputs you need
- **Step 3**: Create a single line to run the function

#You need two input columns x and y, so I will create two columns x and y

x = c(1,2,3,4)

y = c(5,6,7,8)

Then, I run the function

```
>sumtwocolumns(x,y)
[1]  6  8 10 12


>k = sumtwocolumns(x,y)
>k
[1]  6  8 10 12
```

Can you see what is happening here?

```
sum.of.squares <- function(x,y)
 {
x^2 + y^2
}



> d = sum.of.squares(x,y)
> d
[1] 26   40   58   80
>
```

Functions can return more than one item…an example

```
two.returns <- function(x,y)
{
 z=x^2 + y^2
 return(list(z,min(x), max(y)))
}
```

This function returns 3 items

```
> two.returns(x,y)
[[1]]
[1]   26   40   58   80

[[2]]
[1]   1

[[3]]
[1]   8
```

Now lets call the function

Is there an easier way to call the function?

```
> m = two.returns(x,y)
> m[[1]]
[1]   26   40   58   80
> m[[2]]
[1]   1
> m[[3]]
[1]   8
> m[[4]]
Error in m[[4]] : subscript out of bounds
```

Can R differentiate?

YES!! We can use the D() function

Example:

```
> f = expression(-6*x^3 - sqrt(x))
```

Define expression

```
> D(f,'x')
-(6 * (3 * x^2) + 0.5 * x^-0.5)
```

Find first derivative

```
> D(D(f,'x'), 'x')
-(6 * (3 * (2 * x)) + 0.5 * (-0.5 * x^-1.5))
```

Find second derivative

Can R integrate?

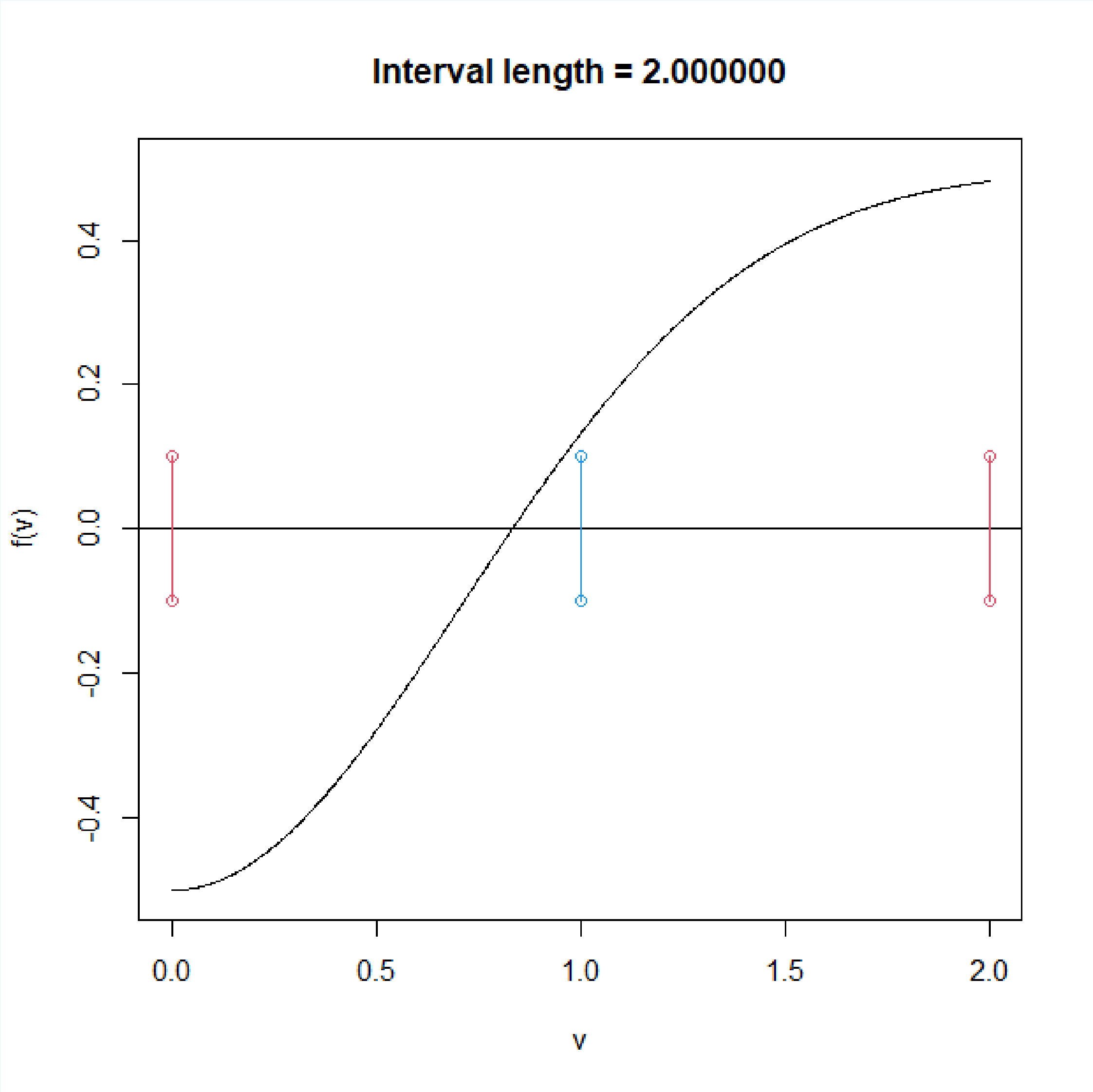
```
> integrand <- function(x) {-6*x^3 - sqrt(x)}
>
> integrate(integrand, lower = .1, upper = .25)
-0.06796086 with absolute error < 7.5e-16
```

Roots of an equation – Plotting the function…

It is always good if we can get a graph of the curve In the code that follows, we try to get a graph of the function f(x) = 1/2 - exp{ -(x^2) }

```
f <- function(x) .5-exp(-(x^2))
v <- seq(0,2,length=1000)
plt <- function(f,a0,b0)
{
  plot(v,f(v),type="l", ylim=c(-.5,.5), xlim=c(0,2),
  main=sprintf("Interval length = %f", (b0-a0)))
  abline(h=0)
  segments(a0,.1,a0,-.1,col=2,lty=1)
  points(a0,.1,col=2)
  points(a0,-.1,col=2)
  segments(b0,.1,b0,-.1,col=2,lty=1)
  points(b0,.1,col=2)
  points(b0,-.1,col=2)
  segments(mean(c(a0,b0)), .1, mean(c(a0,b0)), -.1, col=4, lty=1)
  points(mean(c(a0,b0)),.1,col=4)
  points(mean(c(a0,b0)),-.1,col=4)
}
iter1 <- function(f,I)
{
  m = mean(I)
  if( f(I[1])*f(m) < 0 ) return( c(I[1],m) ) else return( c(m, I[2]) )
}
I <- c(0,2)
f <- function(x) .5-exp(-(x^2))
a0 <- I[1]; b0 <- I[2];
plt(f, a0, b0)
I = iter1(f,I)
```

…The graph that is produced…



The Bisection Method

This is the bisection function. bisect is a function that takes a function f and an interval (a,b) that contains the root followed by a tolerance level.

The Bisection Method

The bisection method requires you to provide a bracket, $c(a_0, b_0)$ as a starting point. The method proceeds by splitting the interval in half, and throws out the interval where the sign did not change, and repeats. Once the interval shrinks below a certain length, δ, it is considered to have converged. Put formally, at step k,

- Calculate $m = (a_{k-1} + b_{k-1})/2$, the midpoint of the current bracket
- if $f(a_{k-1}) \cdot f(m) < 0$ (i.e. the sign does changes over the interval (a_{k-1}, m)), then $a_k = a_{k-1}, b_k = m$.
- otherwise $a_k = m, b_k = b_{k-1}$
- Repeat until $b_k - a_k < \delta$ for some small δ chosen beforehand

The Bisection Method

This is the bisection function. bisect is a function that takes a function f and an interval (a,b) that contains the root followed by a tolerance level.

The Bisection Function

```
bisect <- function(f, a, b, tol)
{
   I <- c(a,b)
   L <- I[2]-I[1]
   while( L > tol )
   {
      m <- mean(I)
      if( f(m)*f(I[1]) < 0 ) I = c(I[1],m) else I = c(m,I[2])
      L <- I[2]-I[1]
   }
   return(mean(I))
}
```

Running the bisection function

Define the function

```
fn <- function(x) 0.5-exp(-(x^2))
```

Use the bisection rule to get am estimate of the root. We need to supply

- the function,
- the interval that contains the root and a guess (starting value).
- The tolerance level.
- The bisect function returns the xvalue.

```
> bisect(fn, 0, 2, 1e-6)
[1] 0.8325543
```

We can get the y value by substituting this xvalue in the function

```
> fn(.8325543)
[1] -2.590558e-07
```

Example 2

In this example, we first generate a random sample of data from a normal distribution with mean 0 and variance 1.
We will assume the variance is known and the mean is unknown. So there is just one parameter to estimate.
This is how we generate the data:

```
X <- rnorm(100)
```

We need the function. Can you see how I got the function below?

```
fn <- function(t) sum(X-t)
```

Find an interval containing the root. We need this interval for the bisection function.

```
> c( fn(-1), fn(1) )
[1]   95.17979 -104.82021
```

Apply the bisect function with tolerance level.

```
bisect(fn, -1, 1, 1e-7)
[1] -0.04820213
```

Recall we can check our answer with the actual MLE which is the sample mean.

```
> mean(X)
[1] -0.0482021
```

So how close are we to what we had expected?

Newton Raphson Method

An extremely fast root-finding approach is Newton's method. This approach is also referred to as *Newton–Raphson iteration*, especially in univariate applications. Suppose that g' is continuously differentiable and that $g''(x^*) \neq 0$. At iteration t, the approach approximates $g'(x^*)$ by the linear Taylor series expansion:

$$0 = g'(x^*) \approx g'(x^{(t)}) + (x^* - x^{(t)})g''(x^{(t)}). \tag{2.8}$$

Since g' is approximated by its tangent line at $x^{(t)}$, it seems sensible to approximate the root of g' by the root of the tangent line. Thus, solving for x^* above, we obtain

$$x^* = x^{(t)} - \frac{g'(x^{(t)})}{g''(x^{(t)})} = x^{(t)} + h^{(t)}. \tag{2.9}$$

This equation describes an approximation to x^* that depends on the current guess $x^{(t)}$ and a refinement $h^{(t)}$. Iterating this strategy yields the updating equation for Newton's method:

$$x^{(t+1)} = x^{(t)} + h^{(t)}, \tag{2.10}$$

Newton Raphson continued

where $h^{(t)} = -g'(x^{(t)}) \,/\, g''(x^{(t)})$. The same update can be motivated by analytically solving for the maximum of the quadratic Taylor series approximation to $g(x^*)$, namely $g(x^{(t)}) + (x^* - x^{(t)})g'(x^{(t)}) + (x^* - x^{(t)})^2 g''(x^{(t)})/2$. When the optimization of g corresponds to an MLE problem where $\hat{\theta}$ is a solution to $l'(\theta) = 0$, the updating equation for Newton's method is

$$\theta^{(t+1)} = \theta^{(t)} - \frac{l'(\theta^{(t)})}{l''(\theta^{(t)})}. \tag{2.11}$$

Newton Raphson Method

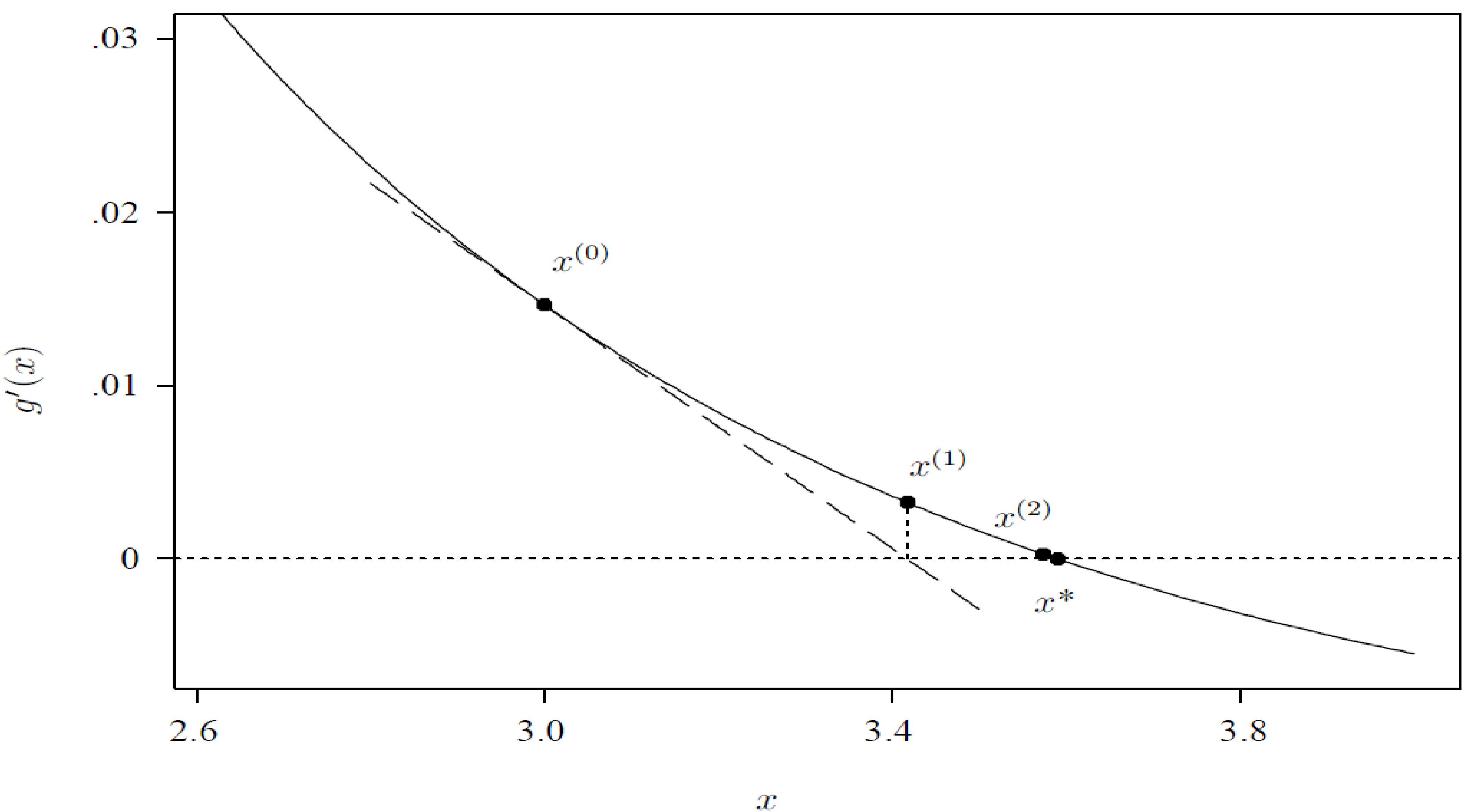


FIGURE 2.3 Illustration of Newton's method applied in Example 2.2. At the first step, Newton's method approximates g' by its tangent line at $x^{(0)}$, whose root $x^{(1)}$ serves as the next approximation of the true root x^*. The next step similarly yields $x^{(2)}$, which is already quite close to x^*.

A simple Newton-Raphson R function Method

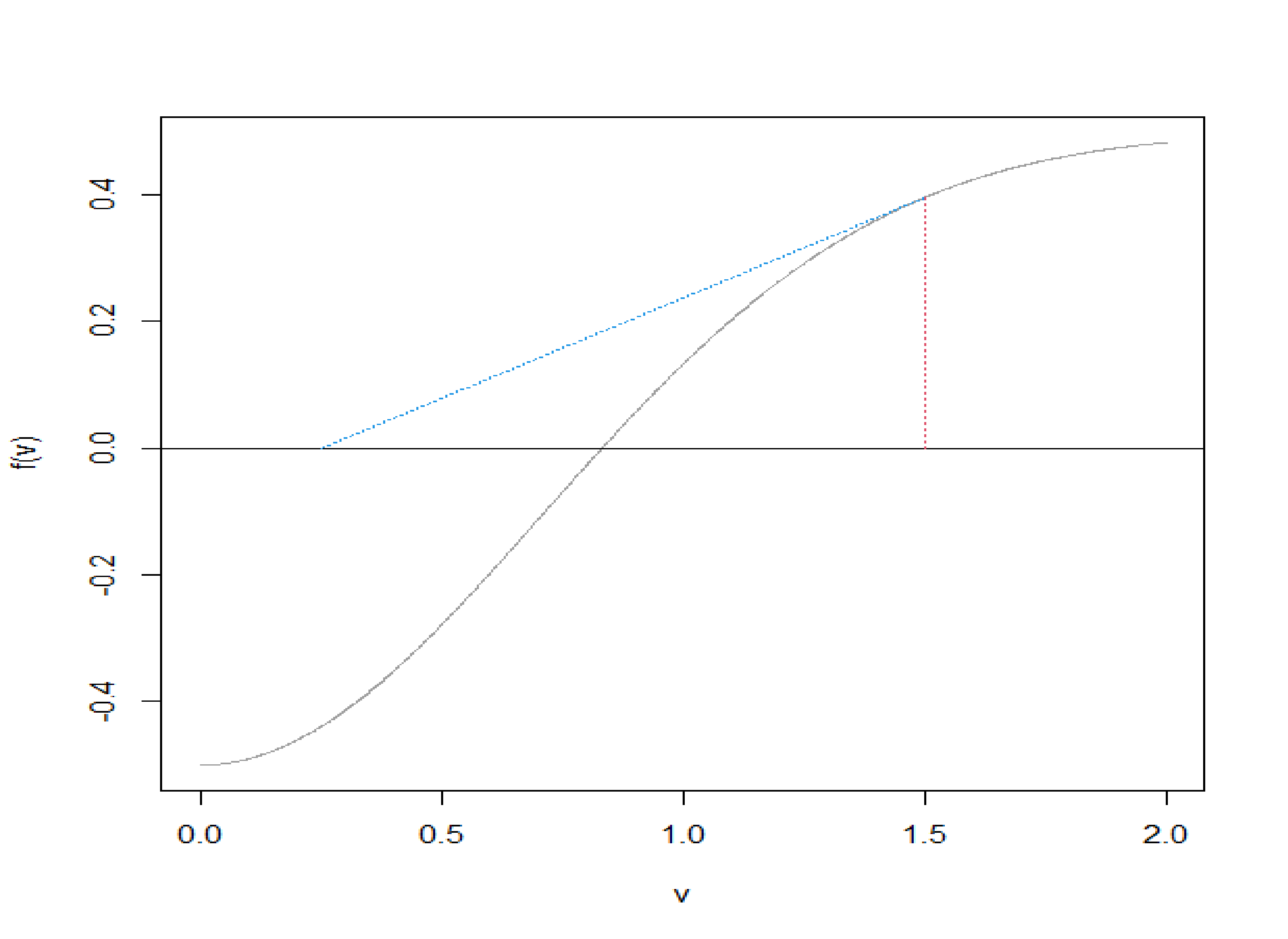
For this method, we need the function fast well as the first derivative df.

```
#This is the function
f <- function(x) .5-exp(-(x^2))

#This is the derivative of the function
df <- function(x) 2*x*exp(-(x^2))

## Lets plot the graph first
v <- seq(0,2,length=1000)
plot(v,f(v),type="l",col=8)
abline(h=0)
x0 <- 1.5
segments(x0,0,x0,f(x0),col=2,lty=3)
slope <- df(x0)
g <- function(x) f(x0) + slope*(x - x0)
v = seq(x0 - f(x0)/df(x0),x0,length=1000)
lines(v,g(v),lty=3,col=4)
```

The plot of the graph



Lets do one iteration of the Newton Raphson before proceeding…

##Starting point

x0 = 1.5

This is our NR update

```
x0 <- x0 - f(x0)/df(x0)
```

```
> x0 <- x0 - f(x0)/df(x0)
>
> x0
[1] 0.252044
```

Now lets use an NR function

```
newton <- function(x0, f, df, d2f, tol=1e-4, pr=FALSE)
{
  k <- 1
  fval <- f(x0)
  grad <- df(x0)
  hess <- d2f(x0)
  xk_1 <- x0
  cond1 <- sqrt( sum(grad^2) )
  cond2 <- Inf
  if( (cond1 < tol) ) return(x0)
  while( (cond1 > tol) & (cond2 > tol) )
  {
    L <- 1
    bool <- TRUE
    while(bool == TRUE)
    {
      xk <- xk_1 - L * solve(hess) %*% grad
      if( f(xk) > fval )
      {
         bool = FALSE
        grad <- df(xk)
        fval <- f(xk)
        hess <- d2f(xk)
      } else
      {
       L = L/2
       if( abs(L) < 1e-20 ) return("Failed to find uphill step - try new start values")
      }
    }
       cond1 <- sqrt( sum(grad^2) )
    cond2 <- sqrt( sum( (xk-xk_1)^2 ))/(tol + sqrt(sum(xk^2)))
    k <- k + 1
    xk_1 <- xk
    }

  if(pr == TRUE) print( sprintf("Took %i iterations", k) )
  return(xk)
}
```

Illustrate the function using a simple function

```
> f   <- function(x) exp(-(x^2))
> df  <- function(x) -2*x*f(x)
> d2f <- function(x) -2*f(x)-2*x*df(x)
> newton(2/3, f, df, d2f, 1e-7)
             [,1]
[1,] -3.233794e-09
```

Now illustrate the NR method with two parameters

This is the approach, we first simulate 100 values from a known distribution with the parameters KNOWN (off course).

We will generate n=100 values from a Normal(2,4). Then we will try to use the NR method to estimate those parameters.

```
X <-  rnorm(100, mean=2, sd=sqrt(4))
n  <- 100
```

- We will need define the likelihood function
- Then we take the logs of that function
- We will see why it is the sum of logs
- It requires a vector t = c(mu, sigma)

Define the sum of the likelihood function

```
f <- function(t)
{
   if( t[2] > 0)
   {
      return( sum( dnorm(X, mean=t[1], sd=sqrt(t[2]), log=TRUE) ) )
   } else
   {
      return(-Inf)
   }
}
```

The vector of score functions – first differentials

```
df <- function(t)
{
mu <- t[1]
sig <- t[2]
 g <- rep(0,2)
 g[1] <- (1/sig) * sum(X - mu)
 g[2] <- (-n/(2*sig)) + sum( (X-mu)^2 )/(2*sig^2)
 return(g)
}
```

The Hessian matrix of second differentials

```
d2f <- function(t)
{
 mu <- t[1]; sig <- t[2];
 h <- matrix(0,2,2)
 h[1,1] <- -n/sig
 h[2,2] <- (n/(2*sig^2)) - sum( (X-mu)^2 )/(sig^3)
 h[1,2] <- -sum( (X-mu) )/(sig^2)
 h[2,1] <- h[1,2]
 return(h)
}
```

Now apply the NR

We simply apply newton's method now.

```
> newton( c(0,1), f,  df,  d2f)
          [,1]
[1,]   2.050354
[2,]   3.151090
```

Lets compare the results with actual MLEs

```
> c( mean(X), (n-1)*var(X)/n)
[1] 2.050354    3.151090
```