Assignment2_Zhuolin

Zhuolin Wang

2/3/2020

Part A: The Art Show

Construct a plot in which you sample start values for Newton-Raphson on a 2D grid. • Color each start point by either: • a) Which root you reach from that start point • b) How many iterations are needed to reach the root you eventually reach form that point.

The two functions I use are Julia Sets one and my own randomly choosen one. Julia Sets one is

$$y = z^2 + c$$

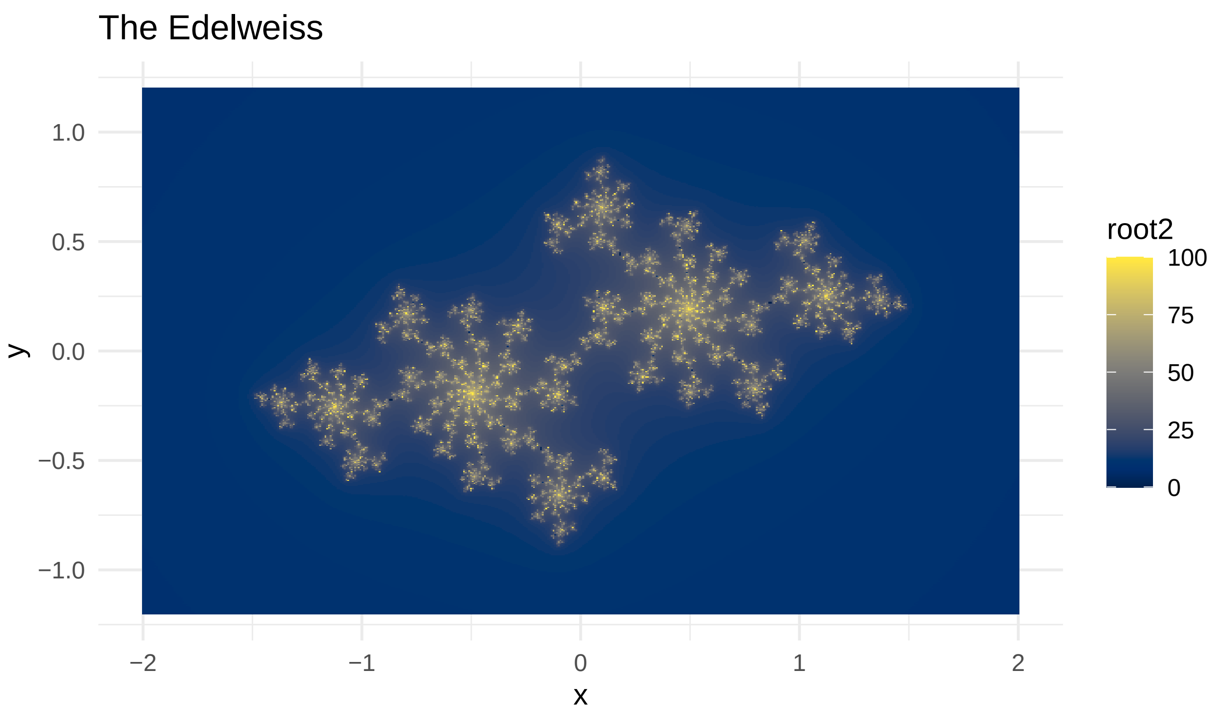
, where c = -0.70176 - 0.3842i(from wikipedia); the other is

$$y = x^8 + 3x^6 - log(8x)$$

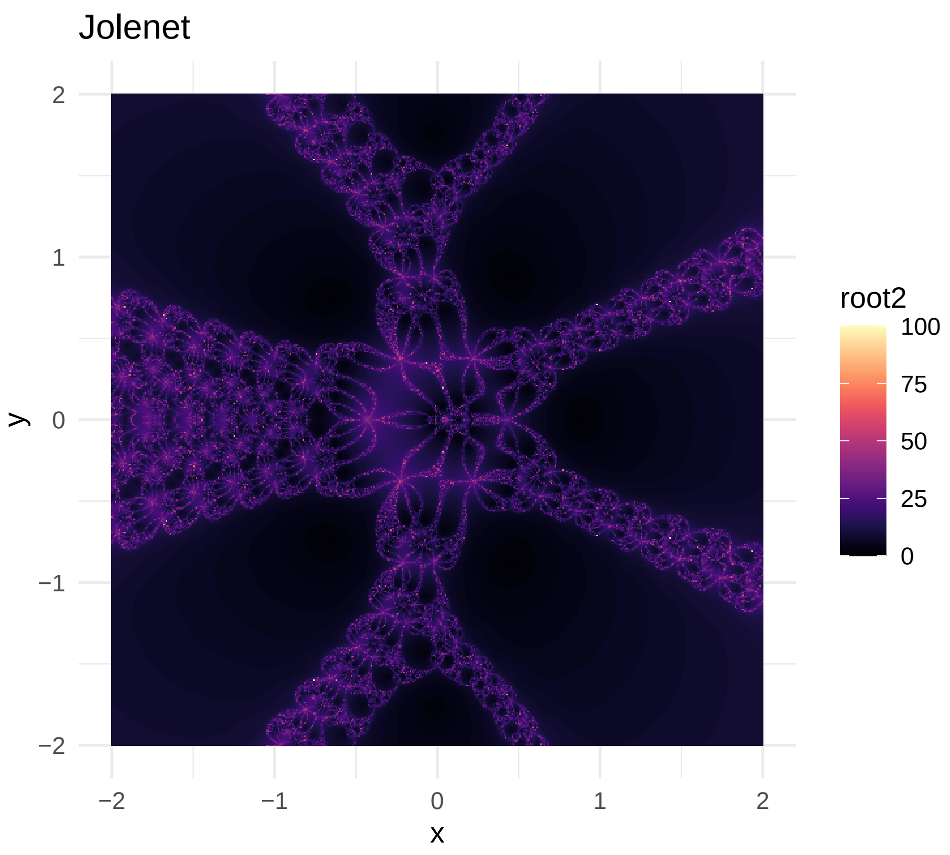
.

```r
plan(multicore)    # this tells it to actually run in parallel
# The implement of the Newton-Raphson method on complex numbers

# using R's built-in 'complex' number definition, which is written so could use +,-,*,/,^ in the usual 

# First define a function to work with - it will return two values.

JuliaSet <- function(z){
  c <- -0.70176 - 0.3842i
  return(z^2 + c)
}

Myownfunc <- function(x){
  return(c(x^8 + 3*x^6 - log(8*x), 8 * x^7 +18*x^5 - 1/(8*x)))
}

# Next we define a global variable to control whether we record roots or the number of iterations taken
# This is an inelegant way of doing it. It should really be part of the argument to the relevent functi
bRootOrIterations <- 0
# Set <-1 to record which root is found, or <- 0 to record number of iterations needed

# Define Newton-Raphson function
NewtonRaphson <- function(func, StartingValue, Tolerance, MaxNumberOfIterations){
  Deviation <- abs(func(StartingValue)[1])
  i <- 0
  x <- StartingValue
  #Set up a while loop until we hit the required target accuracy or the max. number of steps
  while ((i < MaxNumberOfIterations) && (Deviation > Tolerance))
  {
    Z <- func(x)
    # To be safe, check that the function and it's derivative are defined at X (either could be NaN if 
    if ((Z[1] == "NaN") || (Z[2] == "NaN")){
      cat("\nFunction or derivative not defined error.\n")
      break
    }
    x <- x - Z[1]/Z[2]
    Deviation <- abs(func(x)[1])
    i <- i + 1
    #cat(paste("\nIteration ",i,":   X=",x,"  Deviation=",Deviation))
  }
  # output the result
#  if (Deviation < Tolerance){
#    cat(paste("\nFound the root point: ",x, " after ", i, #"iterations"))
#  }else{
#    cat(paste("\nConvergence failure. Deviation: ",Deviation, "after #", i,    "iterations"))}

  if (bRootOrIterations == 1) {
    return(x)
  } else {
    return(c(i, i))
  }
}

TwoDNewtonRaphson <- function(func, StartingValue, Tolerance, MaxNumberOfIterations) {
  i <- 0
  NewZ <- StartingValue
  Deviation = abs(func(StartingValue))

  while ((i < MaxNumberOfIterations) && (Deviation > Tolerance)) {
     Z <- func(NewZ)
    if (is.nan(Z)) {
      break
    }
    NewZ <- func(NewZ)
    Deviation <- abs(NewZ[1])
    i <- i + 1
  }
  if (bRootOrIterations == 1) {
    return(NewZ)
  } else {
    return(c(i, i))
  }
}

# Define a function to check whether two points are close together
CloseTo <- function(x, y){
  # returns 1 if x is close to y
  if (abs(x - y) < 0.1) {
    return(1)
  }else{
    return(0)
  }
}

# And now here's the function that will draw a pretty picture
```

```r
Root_calculator <- function(Funcn, xmin, xmax, xsteps, ymin, ymax, ysteps) {
  x <- seq(xmin, xmax, length.out = xsteps)
  y <- seq(ymin, ymax, length.out = ysteps)

  out_dat <- expand.grid(x = x, y = y)

  ThisZ <- complex(1, out_dat$x, out_dat$y)

  Root <- future_sapply(ThisZ,
                        FUN = NewtonRaphson,
                        func = Funcn,
                        Tolerance = 1e-1,
                        MaxNumberOfIterations = 100)

  if(bRootOrIterations == 0) {
    out_dat$color <- 261 + 5 * Root[1, ]
    out_dat$root1 <- Root[1, ]
    out_dat$root2 <- Root[2, ]
  } else {
    out_dat$color <- 261 + 5 * Root
    out_dat$root1 <- Root
  }

  return(out_dat)
}

Root_calculator2 <- function(Funcn, xmin, xmax, xsteps, ymin, ymax, ysteps) {
  # First define a grid of x and y coordinates over which to run Newton-Raphson.
  # When we run ut for the point (x,y) it will start with the complex number x+iy

  x <- seq(xmin, xmax, length.out = xsteps)
  y <- seq(ymin, ymax, length.out = ysteps)

  out_dat <- expand.grid(x = x, y = y)

  ThisZ <- complex(1, out_dat$x, out_dat$y)

  Root <- future_sapply(ThisZ,
                        FUN = TwoDNewtonRaphson,
                        func = Funcn,
                        Tolerance = 1e-2,
                        MaxNumberOfIterations = 100)

  if(bRootOrIterations == 0) {
    out_dat$color <- 261 + 5 * Root[1, ]
    out_dat$root1 <- Root[1, ]
    out_dat$root2 <- Root[2, ]
  } else {
    out_dat$color <- 261 + 5 * Root
    out_dat$root1 <- Root
  }

  return(out_dat)
}

ggplot_plotter <- function(data) {
  ggplot(data, aes(x, y, fill = root2)) +
    geom_tile() +
    scale_fill_gradientn(colours = magma(256)) +
    theme_minimal() +
    coord_fixed() +
    labs(title = "Jolenet")
}

ggplot_plotter2 <- function(data) {
  ggplot(data, aes(x, y, fill = root2)) +
    geom_tile() +
    scale_fill_gradientn(colours = cividis(256)) +
    theme_minimal() +
    coord_fixed() +
    labs(title = "The Edelweiss")
}

A <- Root_calculator2(JuliaSet, -2, 2, 500, -1.2, 1.2, 500)
ggplot_plotter2(A)
```



```
B <- Root_calculator(Myownfunc, -2, 2, 500, -2, 2, 500)
ggplot_plotter(B)
```



```r
plan(sequential)
```

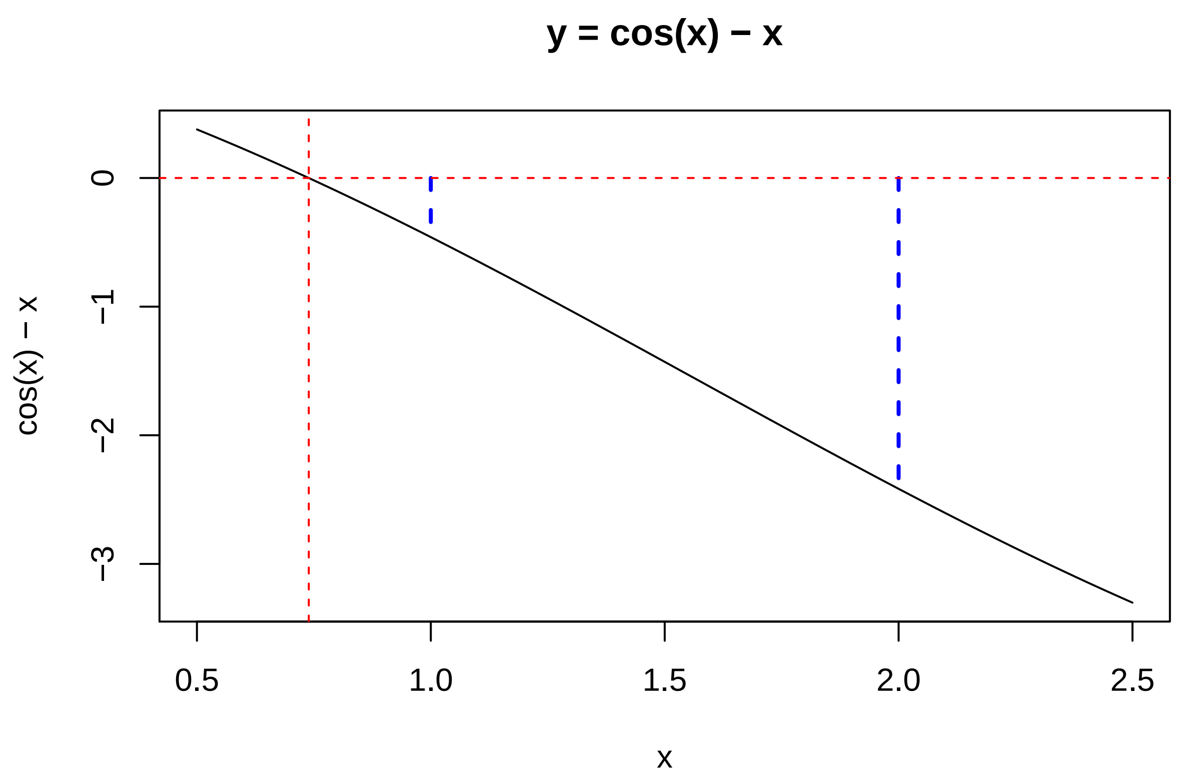
Part B:Secant Method

When thers is a more complex derivative for f(x), Newton method would be less useful. In this situation we could consider using Secant method to do an approximation for the root, if we want to use a way faster than bisection to find the root. Newton Raphson usually faster than Secant method, but if the derivative is hard to find, Secant method could be an alternative choice to find the root.

```r
# Define Secant function
Secant <- function(func, x0, x1, Tolerance, MaxNumberOfIterations){
  i <- 0
  y0 <- func(x0)
  y1 <- func(x1)
  segments(x0, 0, x0, y0, col = "blue", lty = 2, lwd = 2)
  segments(x1, 0, x1, y1, col = "blue", lty = 2, lwd = 2)
  #Set up a while loop until we hit the required target accuracy or the max. number of steps
  while ((i < MaxNumberOfIterations) && (abs(y1) > Tolerance))
  {
    # To be safe, check that the function defined at X (either could be NaN if you are unlucky)
    if ((y0 == "NaN") || (y1 == "NaN")){
      cat("\nFunction not defined error.\n")
      break
    }
    if ((y0 == y1)){
      cat("\nCan not find next x since there is no intersect with zero.\n")
      break
    }
    x2 <- x1 - y1/((y1 - y0)/(x1 - x0))
    y2 <- func(x2)
    x0 <- x1
    x1 <- x2
    y0 <- func(x0)
    y1 <- y2
    i <- i + 1
    cat(paste("\nIteration ",i,":   X=",x1))

  }
  # output the result
  if (abs(y2) < Tolerance){
    cat(paste("\nFound the root point: ",x1, " after ", i, "iterations"))
  }else{
    cat(paste("\nConvergence failure. after ", i,   "iterations"))}
  return(x1)
}

# Define cos(x) - x
start_time <- Sys.time()
F1 <- function(x){
        return(cos(x) - x)
}
# Plot
curve(cos(x) - x, 0.5, 2.5, main = "y = cos(x) - x")
a <- Secant(F1, 1, 2, 1e-3, 40)
```

```
##
## Iteration  1 :   X= 0.765034682391819
## Iteration  2 :   X= 0.742299406864944
## Iteration  3 :   X= 0.739103270158936
## Found the root point:  0.739103270158936  after  3 iterations
```

```
abline(h = 0, v = a, col = "red", lty = 2, lwd = 1)
```



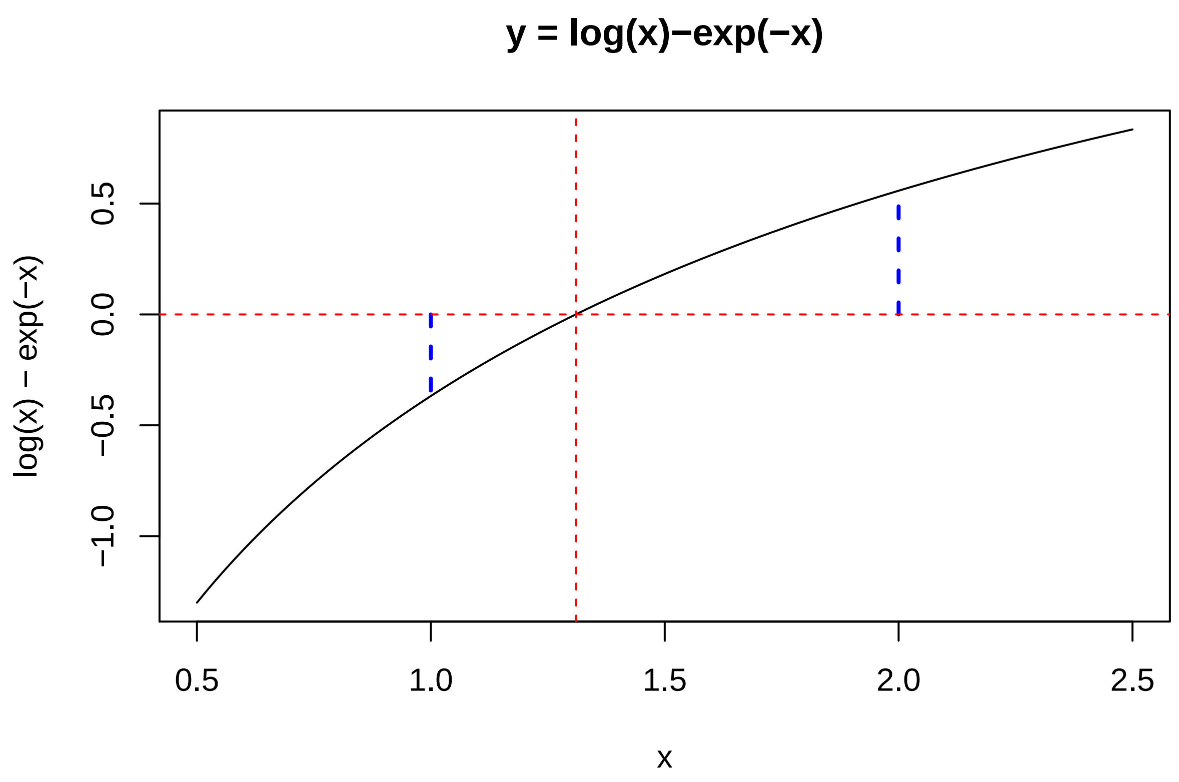
```
end_time <- Sys.time()
cat("\n t=", end_time - start_time)
```

```
##
##  t= 0.03179693
```

```
# Define log(x)-exp(-x)
start_time <- Sys.time()
F2 <- function(x){
        return(log(x) - exp(-x))
}
# Plot
curve(log(x) - exp(-x), 0.5, 2.5, main = "y = log(x)-exp(-x)")
b <- Secant(F2, 1, 2, 1e-3, 40)
```

```
##
## Iteration  1 :   X= 1.39741048216961
## Iteration  2 :   X= 1.28547612015065
## Iteration  3 :   X= 1.31067675808254
## Found the root point:  1.31067675808254  after  3 iterations
```

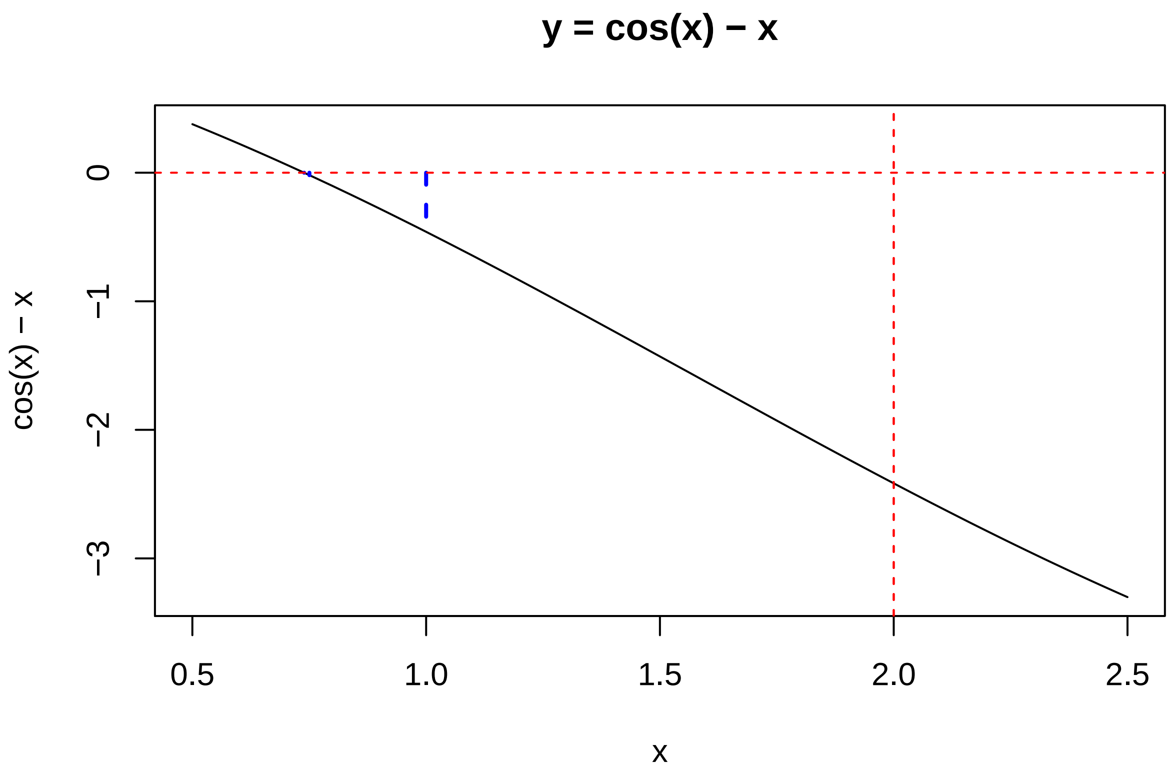
```
abline(h = 0, v = b, col = "red", lty = 2, lwd = 1)
```



```r
end_time <- Sys.time()
cat("\n t=", end_time - start_time)
```

```
##
##  t= 0.008934021
```

```r
# Define Newton-Raphson Function
bRootOrIterations <- 0
NewtonRaphson <- function(func, StartingValue, Tolerance, MaxNumberOfIterations){
  Deviation <- abs(func(StartingValue)[1])
  i <- 0
  x <- StartingValue
  segments(x, 0, x, func(x)[1], col = "blue", lty = 2, lwd = 2)
  while ((i < MaxNumberOfIterations) && (Deviation > Tolerance))
  {
     Z <- func(x)
      if ((Z[1] == "NaN") || (Z[2] == "NaN")){
      cat("\nFunction or derivative not defined error.\n")
      break
    }
    x <- x - Z[1]/Z[2]
    segments(x, 0, x, func(x)[1], col = "blue", lty = 2, lwd = 2)
    Deviation <- abs(func(x)[1])
    i <- i + 1
  }
  if (bRootOrIterations == 1) {
    return(x)
  } else {
    return(c(i, i))
  }
}


# Define cos(x) - x
start_time <- Sys.time()
F3 <- function(x){
        return(c(cos(x) - x, -sin(x) - 1))
}
# Plot
curve(cos(x) - x, 0.5, 2.5, main = "y = cos(x) - x")
c <- NewtonRaphson(F3, 1, 1e-3, 40)
abline(h = 0, v = c, col = "red", lty = 2, lwd = 1)
```

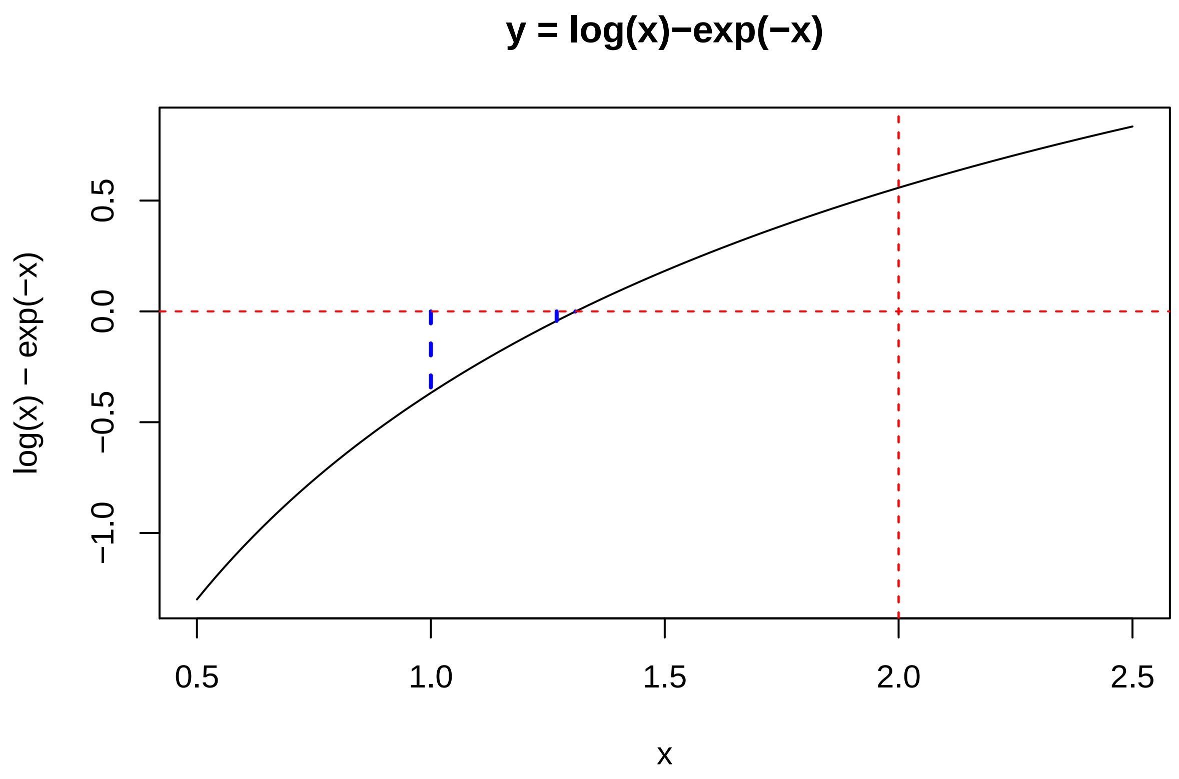


```r
end_time <- Sys.time()
cat("\n t=", end_time - start_time)
```

```
##
##  t= 0.02727509
```

```r
# Define log(x)-exp(-x)
start_time <- Sys.time()
F4 <- function(x){
        return(c(log(x) - exp(-x), 1/x + exp(-x)))
}
# Plot
curve(log(x) - exp(-x), 0.5, 2.5, main = "y = log(x)-exp(-x)")
```

```
b <- NewtonRaphson(F4, 1, 1e-3, 40)
abline(h = 0, v = b, col = "red", lty = 2, lwd = 1)
```



```
end_time <- Sys.time()
cat("\n t=", end_time - start_time)
```

```
##
##  t= 0.01119804
```