Stats 102A - Homework 5 Output

Charles Liu (304804942)

February 28, 2020

"By including this statement, I, Charles Liu, declare that all of the work in this assignment is my own original work. At no time did I look at the code of other students nor did I search for code solutions online. I understand that plagiarism on any single part of this assignment will result in a 0 for the entire assignment and that I will be referred to the dean of students."

"I did discuss ideas related to the homework with Christine Yu for parts 1, 3, and 5, with Leah Skelton for part 5, and looked on Campuswire for part 5. At no point did I show another student my code, nor did I look at another student's code."

Loading Necessary Packages:

```
library(ggplot2)
```

1. An IEEE 754 Mini-Floating Point number [22 points, 2pts each part]

In class, I demonstrated the use of a mini-floating point number system using 8 bits. In my class demo, I used 1 bit for the sign, 3 bits for the exponent, and 4 bits for the mantissa. For this problem, imagine I had used 10 bits - 1 bit for the sign, 4 bits for the exponent, and 5 bits for the mantissa.

```
0 0000 00000 # would now represent the decimal value 0
```

Answer the following questions under this new system.

a. What is the bias that would be used for the exponent? What is the largest positive exponent? What is the most negative exponent?

Formula is [(2^(n-1)) - 1]; for this case n = 3. The exponent bias is 7. The largest positive exponent is +7. The most negative exponent is -6.

b. How would the value 5.5 be stored in this system?

It would be stored as: 0 1001 01100

c. What value would the following bit sequence represent `0 0111 00000`?

The binary `0 0111 00000` = 1

d. What value would the following bit sequence represent `0 0111 00001`? (Express as a fraction and also in decimal.)

The binary `0 0111 00001` = [1 + (1/32)] OR 1.03125

e. What is the smallest positive normalized value that can be expressed? (Fill in the bits. Express as a fraction and also in decimal.)

```
0 0001 00000
```

The smallest bit (normalized) would be: 1/64 OR 0.015625 OR 2ˆ-6

f. What is the smallest positive (denormalized) non-zero value that can be expressed? (Fill in the bits. Express as a fraction and also in decimal.)

```
0 0000 00001
```

The smallest bit (denormalized) would be: 1/2048 OR 0.00048828 OR 2ˆ-11

g. What is the largest denormalized value that can be expressed? (Fill in the bits. Express as a fraction and also in decimal.)

```
0 0000 11111
```

The largest bit (denormalized) would be: 31/2048 or 0.01513672

h. What is the largest finite value that can be expressed with this sytem? (Fill in the bits. Express as a fraction and also in decimal.)

```
0 1110 11111
```

The largest bit (finite) would be: 252

i. With our 10 bit floating point system, what is the smallest value you can add to 1 so that the sum will be different from 1? That is, what is the smallest value of x, so that the following will return FALSE? In other words, what is the machine epsilon of this system?

(1/32) is our machince epsilon.

j. What is the smallest value you can add to the number 2 so that the sum will be different from 2? (Express as a fraction)

The smallest value you can add to the number 2 is: (1/16)

k. What is the smallest value you can add to the number 4 so that the sum will be different from 4? (Express as a fraction)

The smallest value you can add to the number 2 is: (1/8)

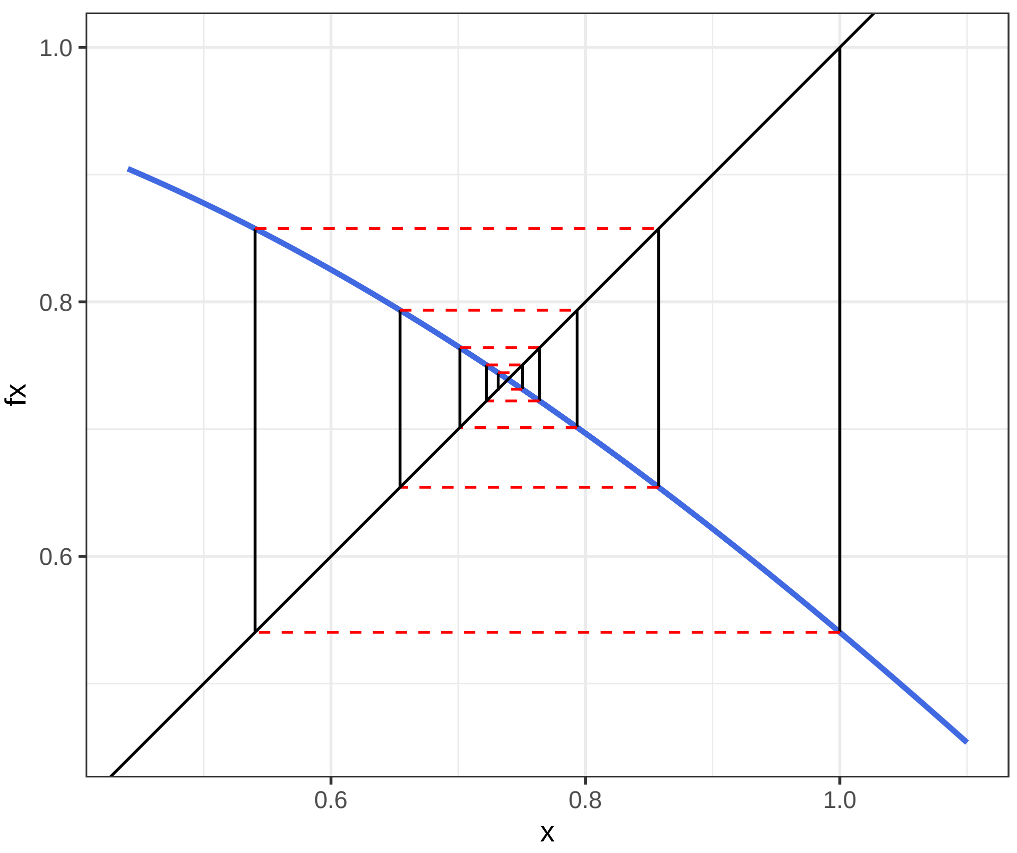
2. Root Finding with Fixed Point Iteration [12 points, 2 points each part]

```r
fixedpoint_show <- function(ftn, x0, iter = 5){
  # applies fixed-point method to find x such that ftn(x) = x
  # ftn is a user-defined function
  # df_points_1 and df_points_2 are used to track each update
  # it will be used to plot the line segments showing each update
  # each line segment connects the points (x1, y1) to (x2, y2)
  df_points_1 <- data.frame(
    x1 = numeric(0),
    y1 = numeric(0),
    x2 = numeric(0),
    y2 = numeric(0))
  df_points_2 <- df_points_1
  xnew <- x0
  cat("Starting value is:", xnew, "\n")
  # iterate the fixed point algorithm
  for (i in 1:iter) {
    xold <- xnew
    xnew <- ftn(xold)
    cat("Next value of x is:", xnew, "\n")
    # vertical line segments, where x1 = x2
    df_points_1[i, ] <- c(x1 = xold, y1 = xold, x2 = xold, y2 = xnew)
    # horizontal line segments, where y1 = y2
    df_points_2[i, ] <- c(x1 = xold, y1 = xnew, x2 = xnew, y2 = xnew)
  }
  # use ggplot to plot the function and the segments for each iteration
  # determine the limits to use for the plot
  # start is the min of these values. we subtract .1 to provide a small margin
  plot_start <- min(df_points_1$x1, df_points_1$x2, x0) - 0.1
  # end is the max of these values
  plot_end <- max(df_points_1$x1, df_points_1$x2, x0) + 0.1
  # calculate the value of the funtion fx for all x
  x <- seq(plot_start, plot_end, length.out = 200)
  fx <- rep(NA, length(x))
  for (i in seq_along(x)) {
    fx[i] <- ftn(x[i])
  }
  function_data <- data.frame(x, fx) # data frame containing the function values
  p <- ggplot(function_data, aes(x = x, y = fx)) +
    geom_line(color = "royalblue", size = 1) + # plot the function
    geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),
                 data = df_points_1, color = "black", lty = 1) +
    geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),
                 data = df_points_2, color = "red", lty = 2) +
    geom_abline(intercept = 0, slope = 1) + # plot the line y = x
    coord_equal() + theme_bw()
  print(p) # produce the plot
  xnew # value that gets returned
}
```

Do part (a) using x0 = 1

```r
### x0 = 1
a2_1 <- function(x) cos(x)
fixedpoint_show(a2_1, 1, iter = 10)
```

```
## Starting value is: 1
## Next value of x is: 0.5403023
## Next value of x is: 0.8575532
## Next value of x is: 0.6542898
## Next value of x is: 0.7934804
## Next value of x is: 0.7013688
## Next value of x is: 0.7639597
## Next value of x is: 0.7221024
## Next value of x is: 0.7504178
## Next value of x is: 0.731404
## Next value of x is: 0.7442374
```

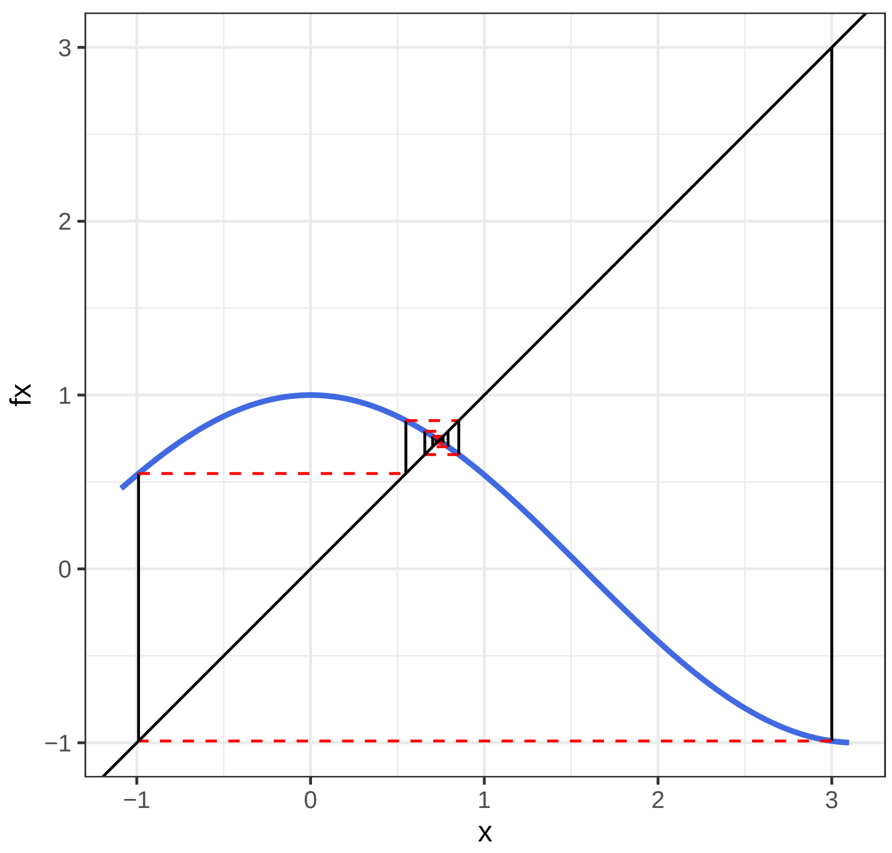


```
## [1] 0.7442374
```

Do part (a) using x0 = 3

```
### x0 = 3
a2_2 <- function(x) cos(x)
fixedpoint_show(a2_2, 3, iter = 10)
```

```
## Starting value is: 3
## Next value of x is: -0.9899925
## Next value of x is: 0.5486961
## Next value of x is: 0.8532053
## Next value of x is: 0.6575717
## Next value of x is: 0.7914787
## Next value of x is: 0.7027941
## Next value of x is: 0.7630392
## Next value of x is: 0.7227389
## Next value of x is: 0.7499969
## Next value of x is: 0.731691
```

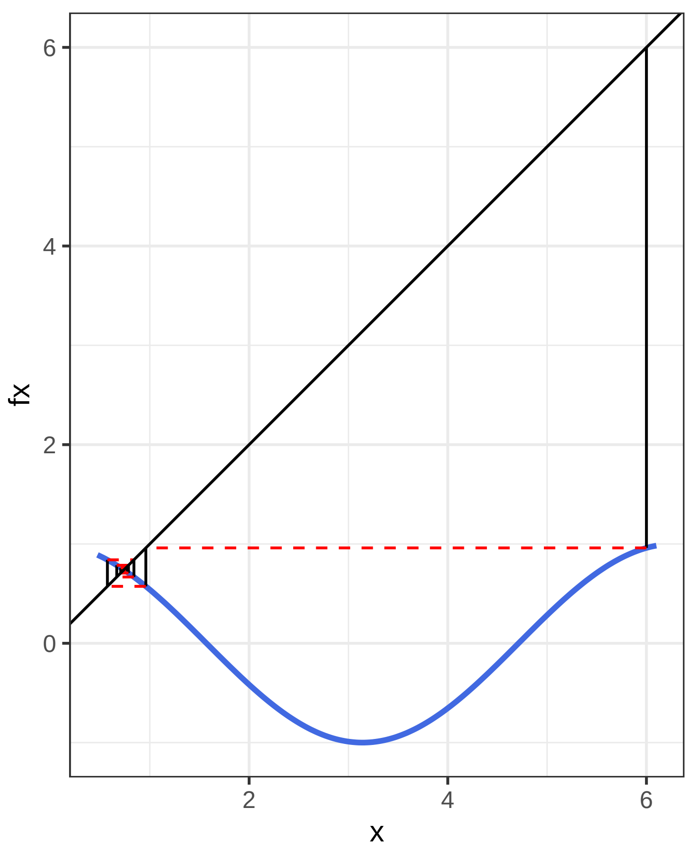


```
## [1] 0.731691
```

Do part (a) using x0 = 6

```
### x0 = 6
a2_3 <- function(x) cos(x)
fixedpoint_show(a2_3, 6, iter = 10)
```

```
## Starting value is: 6
## Next value of x is: 0.9601703
## Next value of x is: 0.5733805
## Next value of x is: 0.840072
## Next value of x is: 0.6674092
## Next value of x is: 0.7854279
## Next value of x is: 0.7070858
## Next value of x is: 0.7602582
## Next value of x is: 0.7246581
## Next value of x is: 0.7487261
## Next value of x is: 0.7325566
```

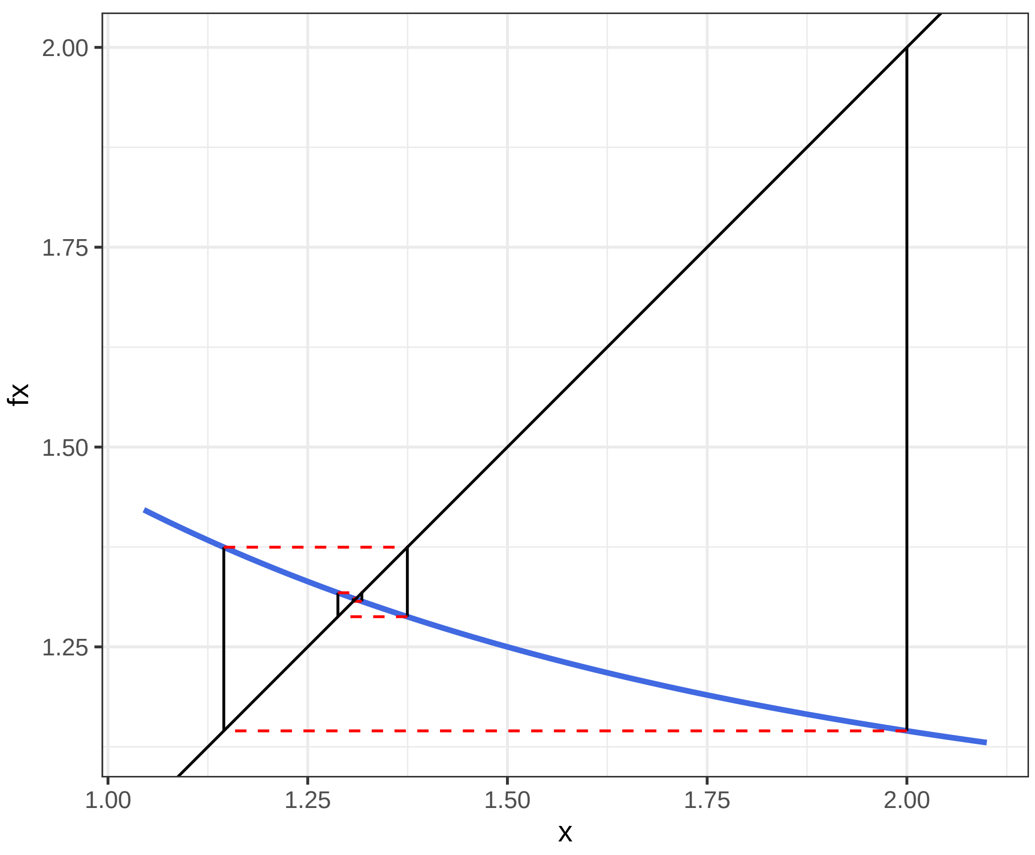


```
## [1] 0.7325566
```

Do part (b) using x0 = 2

```
### x0 = 2
b2_1 <- function(x) exp(exp(-x))
fixedpoint_show(b2_1, 2, iter = 10)
```

```
## Starting value is: 2
## Next value of x is: 1.144921
## Next value of x is: 1.374719
## Next value of x is: 1.287768
## Next value of x is: 1.317697
## Next value of x is: 1.307022
## Next value of x is: 1.310783
## Next value of x is: 1.309452
## Next value of x is: 1.309922
## Next value of x is: 1.309756
## Next value of x is: 1.309815
```

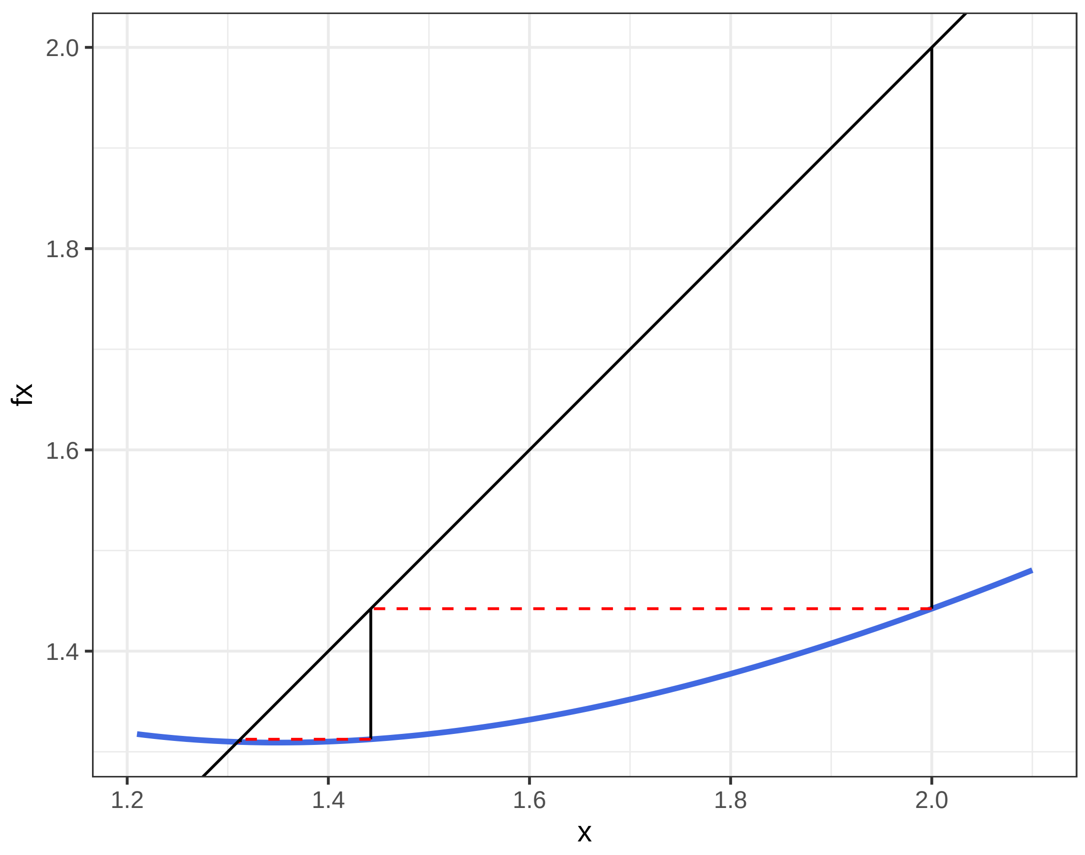


```
## [1] 1.309815
```

Do part (c) using x0 = 2

```r
### x0 = 2
c2_1 <- function(x) x - log(x) + exp(-x)
fixedpoint_show(c2_1, 2, iter = 10)
```

```
## Starting value is: 2
## Next value of x is: 1.442188
## Next value of x is: 1.312437
## Next value of x is: 1.309715
## Next value of x is: 1.309802
## Next value of x is: 1.309799
## Next value of x is: 1.3098
## Next value of x is: 1.3098
## Next value of x is: 1.3098
## Next value of x is: 1.3098
## Next value of x is: 1.3098
```

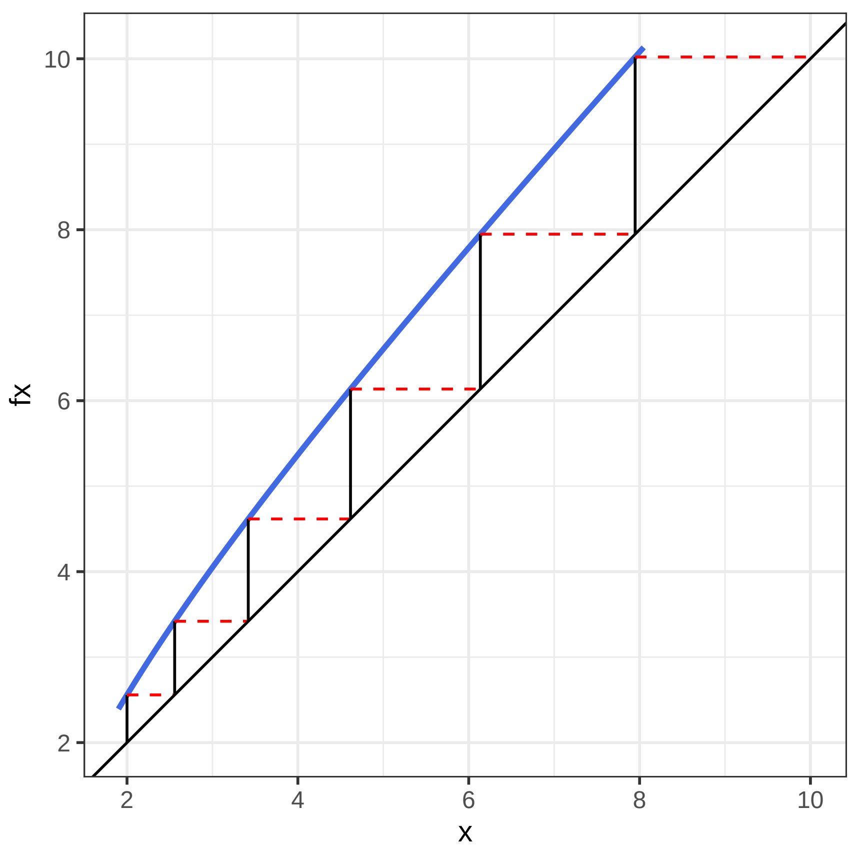


```
## [1] 1.3098
```

Do part (d) using x0 = 2, no more than 6 iterations

```
### x0 = 2
d2_1 <- function(x) x + log(x) - exp(-x)
fixedpoint_show(d2_1, 2, iter = 6)
```

```
## Starting value is: 2
## Next value of x is: 2.557812
## Next value of x is: 3.41949
## Next value of x is: 4.616252
## Next value of x is: 6.135946
## Next value of x is: 7.947946
## Next value of x is: 10.02051
```



```
## [1] 10.02051
```

3. Root Finding with Newton Raphson [22 points, 10 points for completing the code. 1 pts each graph]

```
newtonraphson_show <- function(ftn, x0, iter = 5) {
  # applies Newton-Raphson to find x such that ftn(x)[1] == 0
  # ftn is a function of x. it returns two values, f(x) and f'(x)
  # x0 is the starting point
  # df_points_1 and df_points_2 are used to track each update
  df_points_1 <- data.frame(
    x1 = numeric(0),
    y1 = numeric(0),
    x2 = numeric(0),
    y2 = numeric(0))
  df_points_2 <- df_points_1
  xnew <- x0
  cat("Starting value is:", xnew, "\n")
  # the algorithm
  for(i in 1:iter){
    xold <- xnew
    f_xold <- ftn(xold)
    xnew <- xold - f_xold[1]/f_xold[2]
    cat("Next x value:", xnew, "\n")
    # the line segments. You will need to replace the NAs with the appropriate values
    df_points_1[i, ] <- c(xold, ftn(xold)[1], xold, 0) # vertical segment
```

```r
    df_points_2[i, ] <- c(xold, ftn(xold)[1], xnew, 0) # tangent segment
  }
  plot_start <- min(df_points_1$x1, df_points_1$x2) - 0.1
  plot_end <- max(df_points_2$x1, df_points_2$x2) + 0.1
  x <- seq(plot_start, plot_end, length.out = 200)
  fx <- rep(NA, length(x))
  for (i in seq_along(x)) {
    fx[i] <- ftn(x[i])[1]
  }
  function_data <- data.frame(x, fx)
  p <- ggplot(function_data, aes(x = x, y = fx)) +
    geom_line(color = "royalblue", size = 1) + # plot the function
    geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),
                 data = df_points_1, color = "black", lty = 1) +
    geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),
                 data = df_points_2, color = "red", lty = 2) +
    geom_abline(intercept = 0, slope = 1) # plot the line y = x
  print(p)
  xnew # value that gets returned
}
```

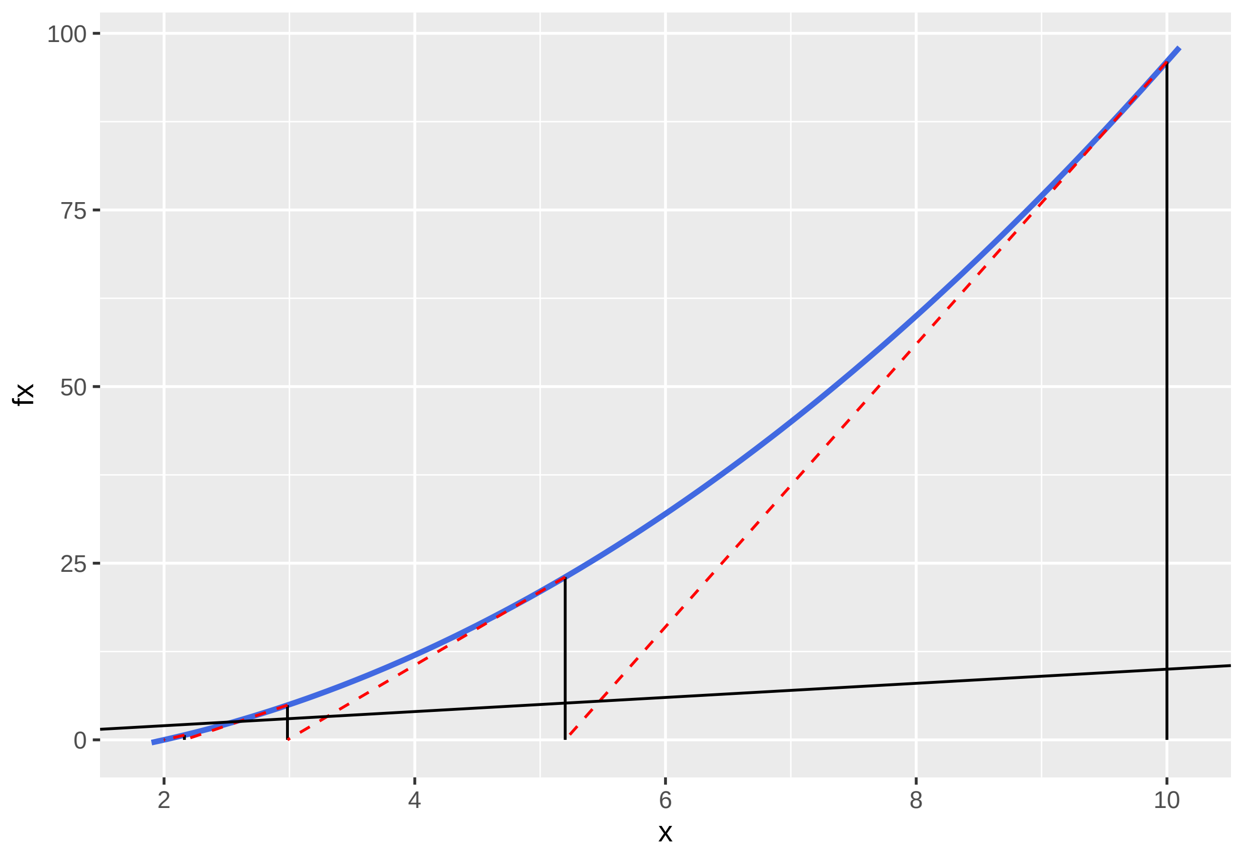
Produce graphs for:

The function f(x) = x^2 - 4 using x0 = 10

```r
f3 <- function(x) {
  value <- x^2 - 4 # f(x)
  derivative <- 2*x # f'(x)
  return(c(value, derivative))
}

### x0 = 10
newtonraphson_show(f3, 10, iter = 8)
```

```
## Starting value is: 10
## Next x value: 5.2
## Next x value: 2.984615
## Next x value: 2.162411
## Next x value: 2.006099
## Next x value: 2.000009
## Next x value: 2
## Next x value: 2
## Next x value: 2
```



```
## [1] 2
```

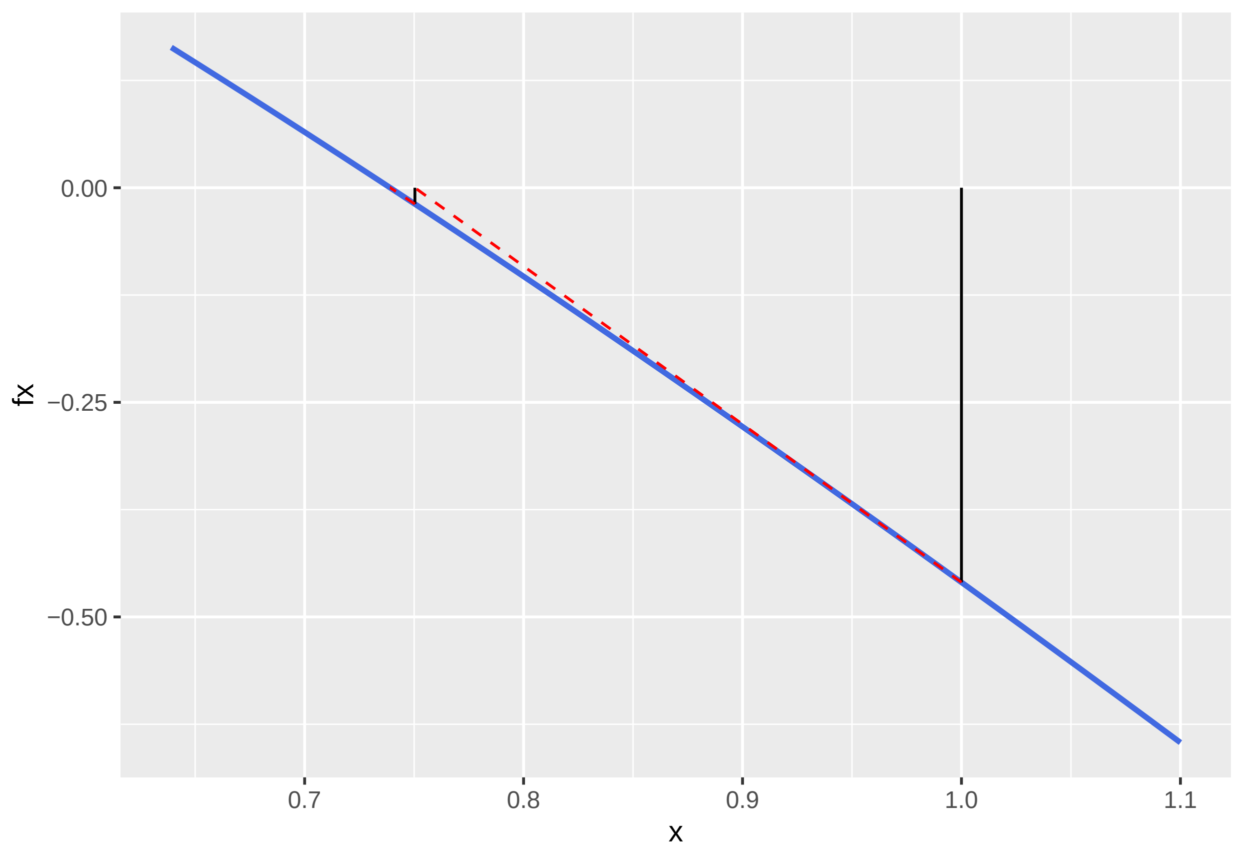
part (a) using x0 = 1, 3, 6 Results should be similar to finding fixed point of cos(x)

Do part (a) using x0 = 1

```
a3_1 <- function(x) {
  value <- cos(x) - x # f(x)
  derivative <- -sin(x) - 1 # f'(x)
  return(c(value, derivative))
}

### x0 = 1
newtonraphson_show(a3_1, 1, iter = 8)
```

```
## Starting value is: 1
## Next x value: 0.7503639
## Next x value: 0.7391129
## Next x value: 0.7390851
## Next x value: 0.7390851
## Next x value: 0.7390851
## Next x value: 0.7390851
## Next x value: 0.7390851
## Next x value: 0.7390851
```



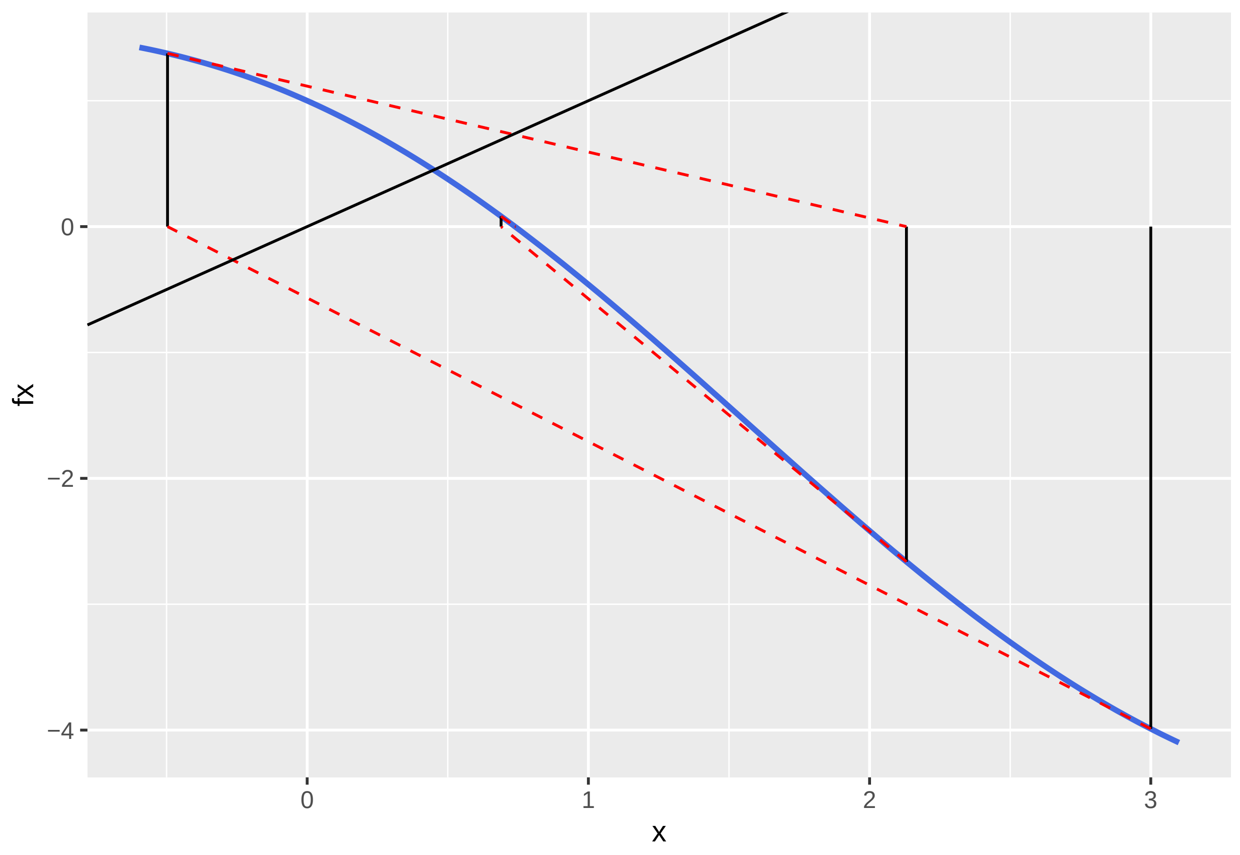
```
## [1] 0.7390851
```

Do part (a) using x0 = 3

```
a3_1 <- function(x) {
  value <- cos(x) - x # f(x)
  derivative <- -sin(x) - 1 # f'(x)
  return(c(value, derivative))
}

### x0 = 3
newtonraphson_show(a3_1, 3, iter = 8)
```

```
## Starting value is: 3
## Next x value: -0.4965582
## Next x value: 2.131004
## Next x value: 0.6896627
## Next x value: 0.739653
## Next x value: 0.7390852
## Next x value: 0.7390851
## Next x value: 0.7390851
## Next x value: 0.7390851
```



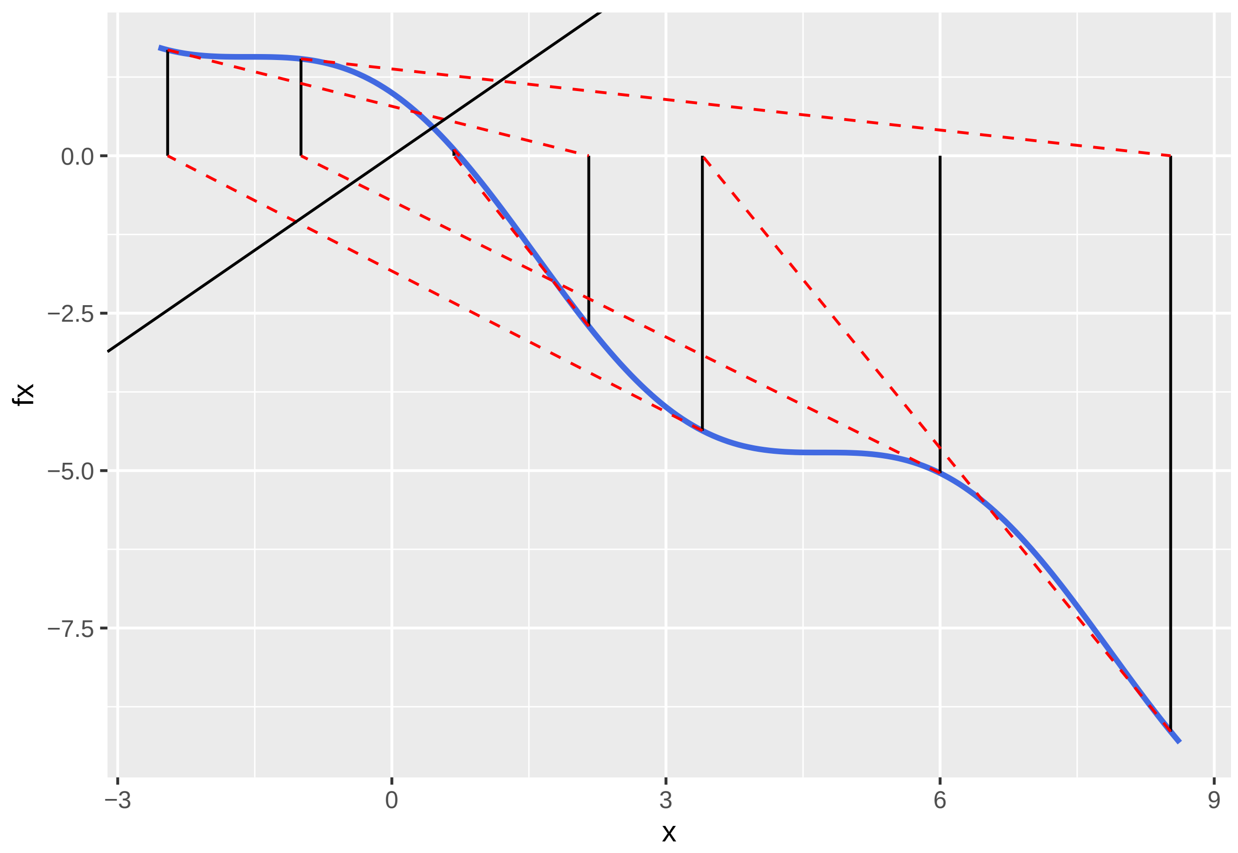
```
## [1] 0.7390851
```

Do part (a) using x0 = 6

```
a3_1 <- function(x) {
  value <- cos(x) - x # f(x)
  derivative <- -sin(x) - 1 # f'(x)
  return(c(value, derivative))
}

### x0 = 6
newtonraphson_show(a3_1, 6, iter = 8)
```

```
## Starting value is: 6
## Next x value: -0.9940856
## Next x value: 8.523426
## Next x value: 3.398358
## Next x value: -2.45325
## Next x value: 2.155349
## Next x value: 0.6792118
## Next x value: 0.7399276
## Next x value: 0.7390853
```



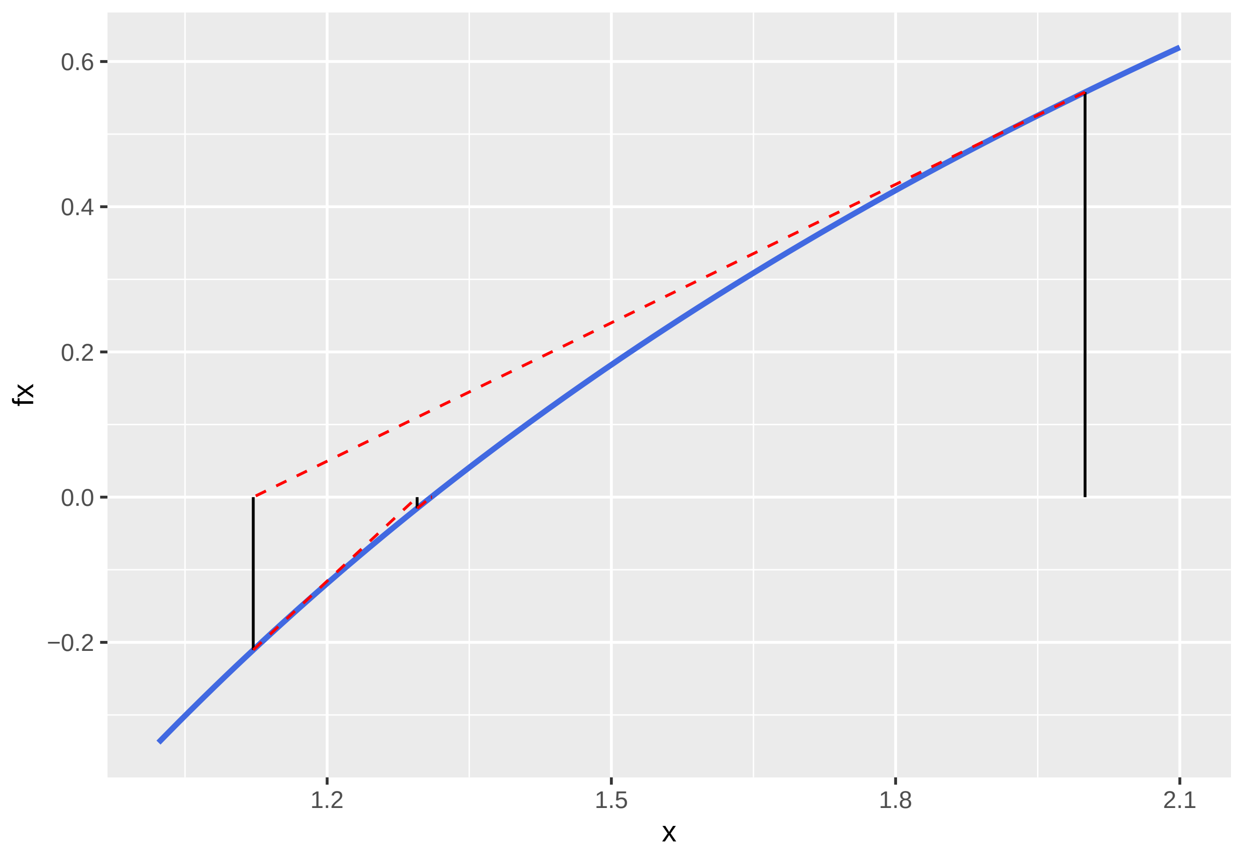
```
## [1] 0.7390853
```

part (b) using x0 = 2 Results should be similar to finding fixed point of exp(exp(-x))

```
b3_1 <- function(x) {
  value <- log(x) - exp(-x)
  derivative <- 1/x + exp(-x)
  return(c(value, derivative))
}

### x0 = 2
newtonraphson_show(b3_1, 2, iter = 8)
```

```
## Starting value is: 2
## Next x value: 1.12202
## Next x value: 1.294997
## Next x value: 1.309709
## Next x value: 1.3098
## Next x value: 1.3098
## Next x value: 1.3098
## Next x value: 1.3098
## Next x value: 1.3098
```



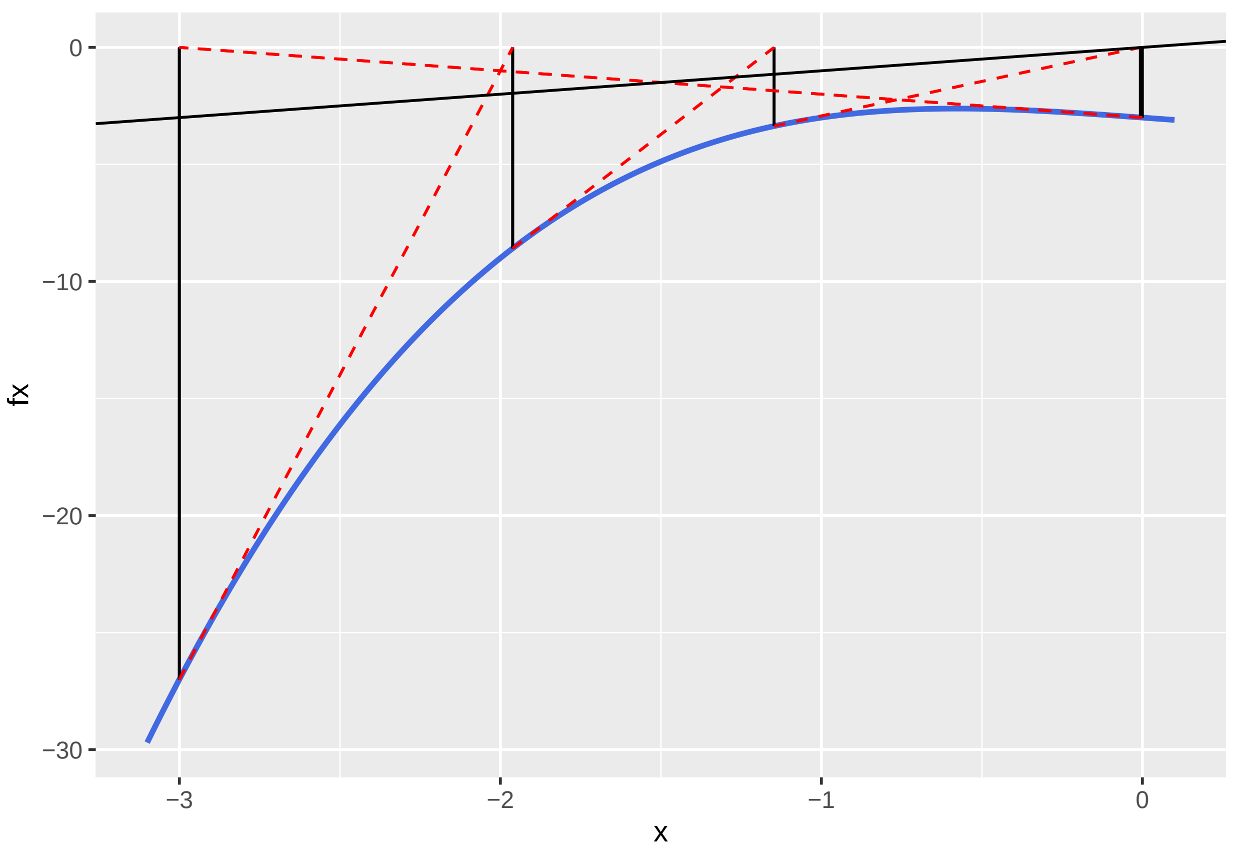
```
## [1] 1.3098
```

part (c) using x0 = 0

```
c3_1 <- function(x) {
  value <- x^3 - x - 3
  derivative <- 3*x^2 - 1
  return(c(value, derivative))
}

### x0 = 0
newtonraphson_show(c3_1, 0, iter = 8)
```

```
## Starting value is: 0
## Next x value: -3
## Next x value: -1.961538
## Next x value: -1.147176
## Next x value: -0.006579371
## Next x value: -3.000389
## Next x value: -1.961818
## Next x value: -1.14743
## Next x value: -0.007256248
```



```
## [1] -0.007256248
```

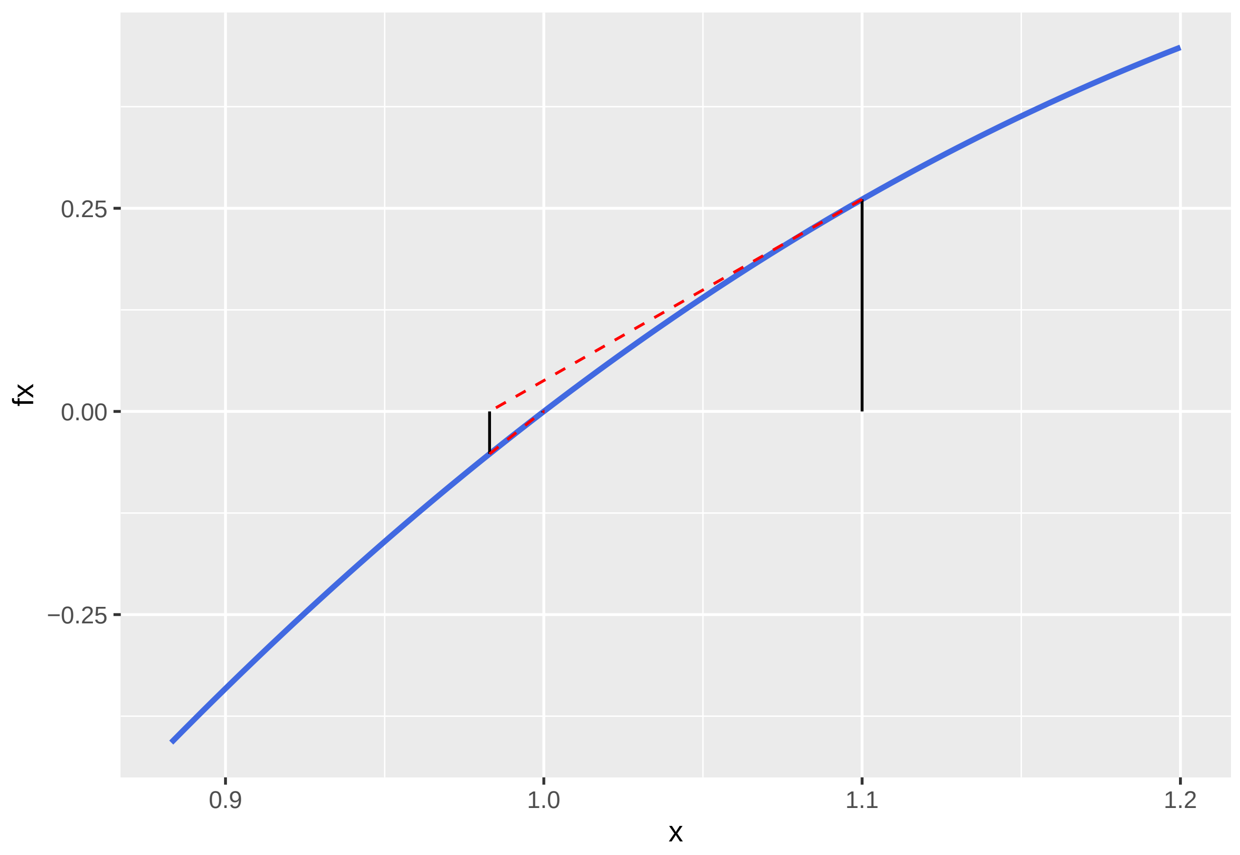
Part (d) using x0 = 1.1, 1.3, 1.4, 1.5, 1.6, 1.7 (should be simple. just repeat the command several times)

Part (d) using x0 = 1.1

```
d3_1 <- function(x) {
  value <- x^3 - 7*x^2 + 14*x - 8
  derivative <- 3*x^2 - 14*x + 14
  return(c(value, derivative))
}

### x0 = 1.1
newtonraphson_show(d3_1, 1.1, iter = 8)
```

```
## Starting value is: 1.1
## Next x value: 0.9829596
## Next x value: 0.9996266
## Next x value: 0.9999998
## Next x value: 1
## Next x value: 1
## Next x value: 1
## Next x value: 1
## Next x value: 1
```



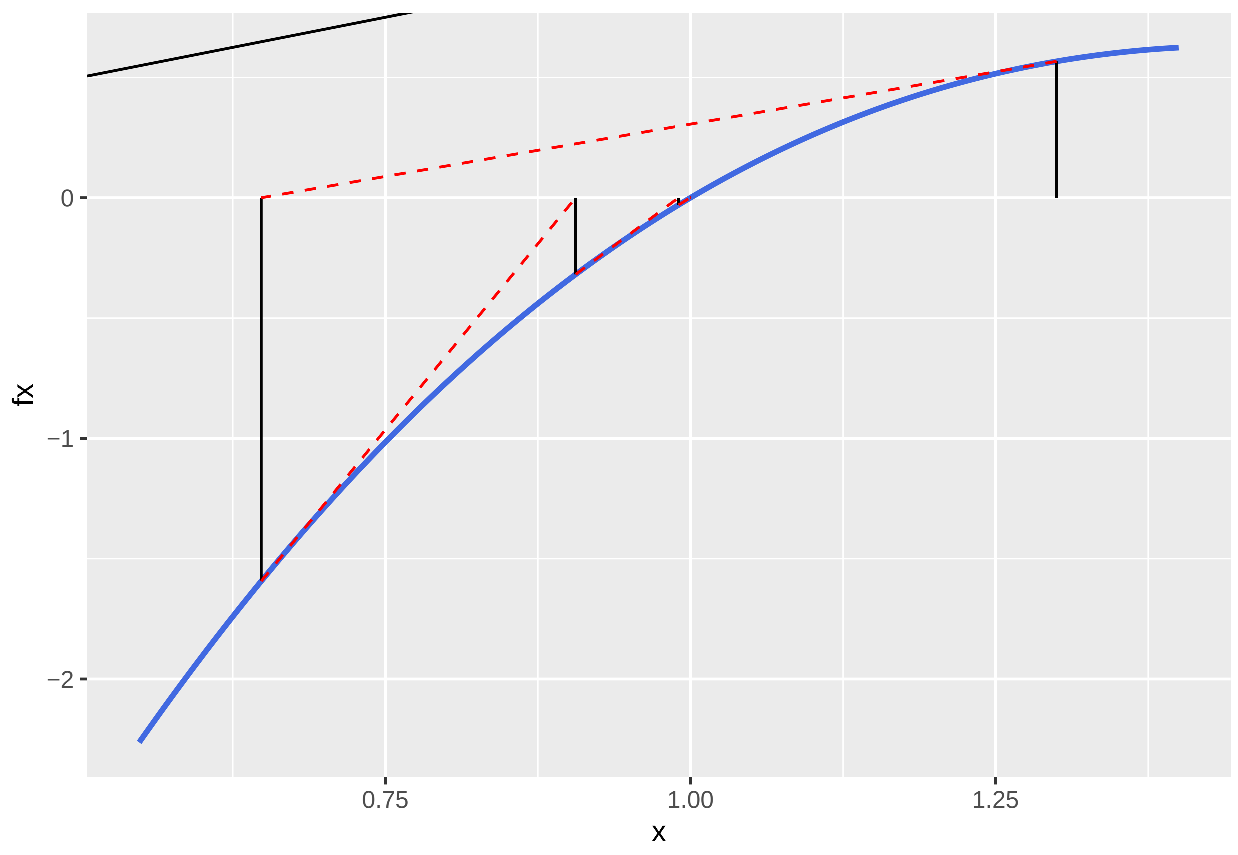
```
## [1] 1
```

Part (d) using x0 = 1.3

```
d3_1 <- function(x) {
  value <- x^3 - 7*x^2 + 14*x - 8
  derivative <- 3*x^2 - 14*x + 14
  return(c(value, derivative))
}

### x0 = 1.3
newtonraphson_show(d3_1, 1.3, iter = 8)
```

```
## Starting value is: 1.3
## Next x value: 0.6482759
## Next x value: 0.9059224
## Next x value: 0.9901916
## Next x value: 0.9998744
## Next x value: 1
## Next x value: 1
## Next x value: 1
## Next x value: 1
```



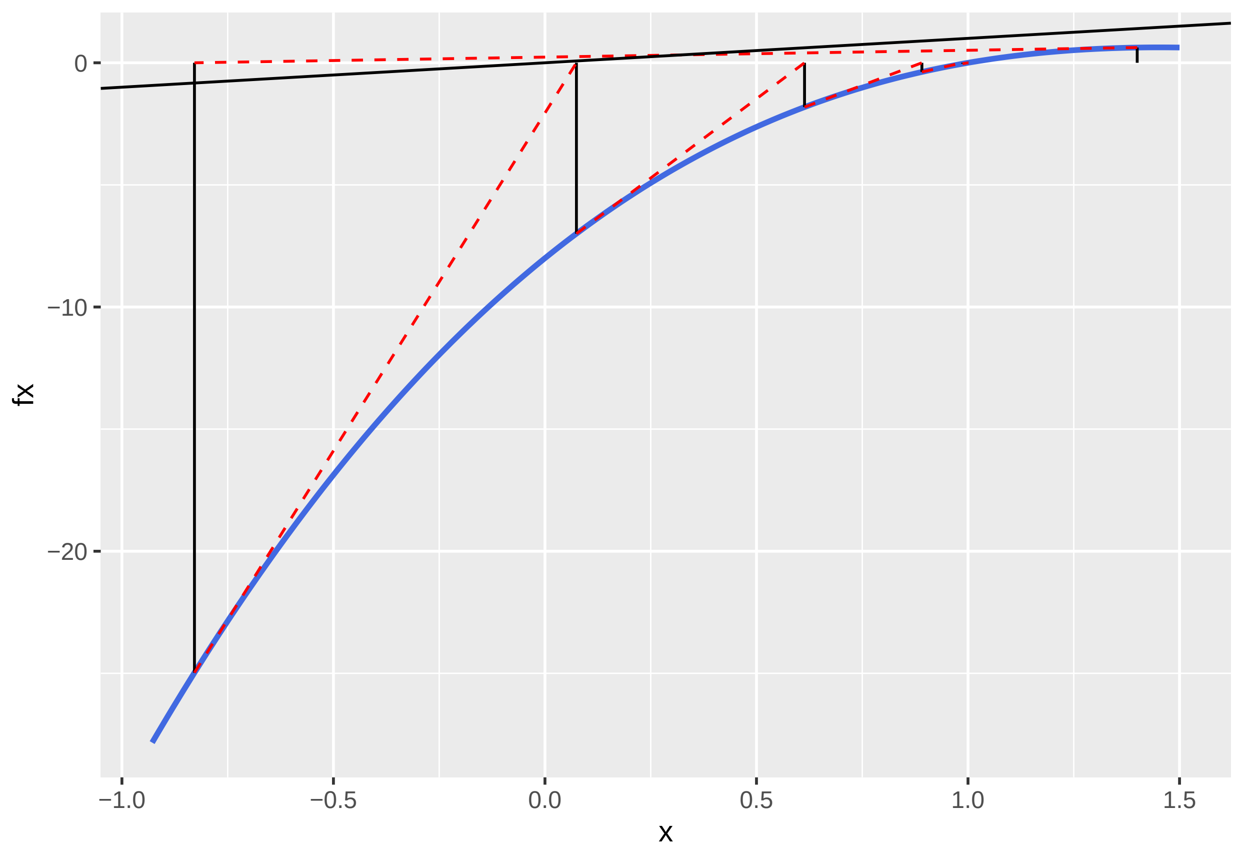
```
## [1] 1
```

Part (d) using x0 = 1.4

```
d3_1 <- function(x) {
  value <- x^3 - 7*x^2 + 14*x - 8
  derivative <- 3*x^2 - 14*x + 14
  return(c(value, derivative))
}

### x0 = 1.4
newtonraphson_show(d3_1, 1.4, iter = 8)
```

```
## Starting value is: 1.4
## Next x value: -0.8285714
## Next x value: 0.07435419
## Next x value: 0.6136214
## Next x value: 0.891034
## Next x value: 0.9871826
## Next x value: 0.9997869
## Next x value: 0.9999999
## Next x value: 1
```



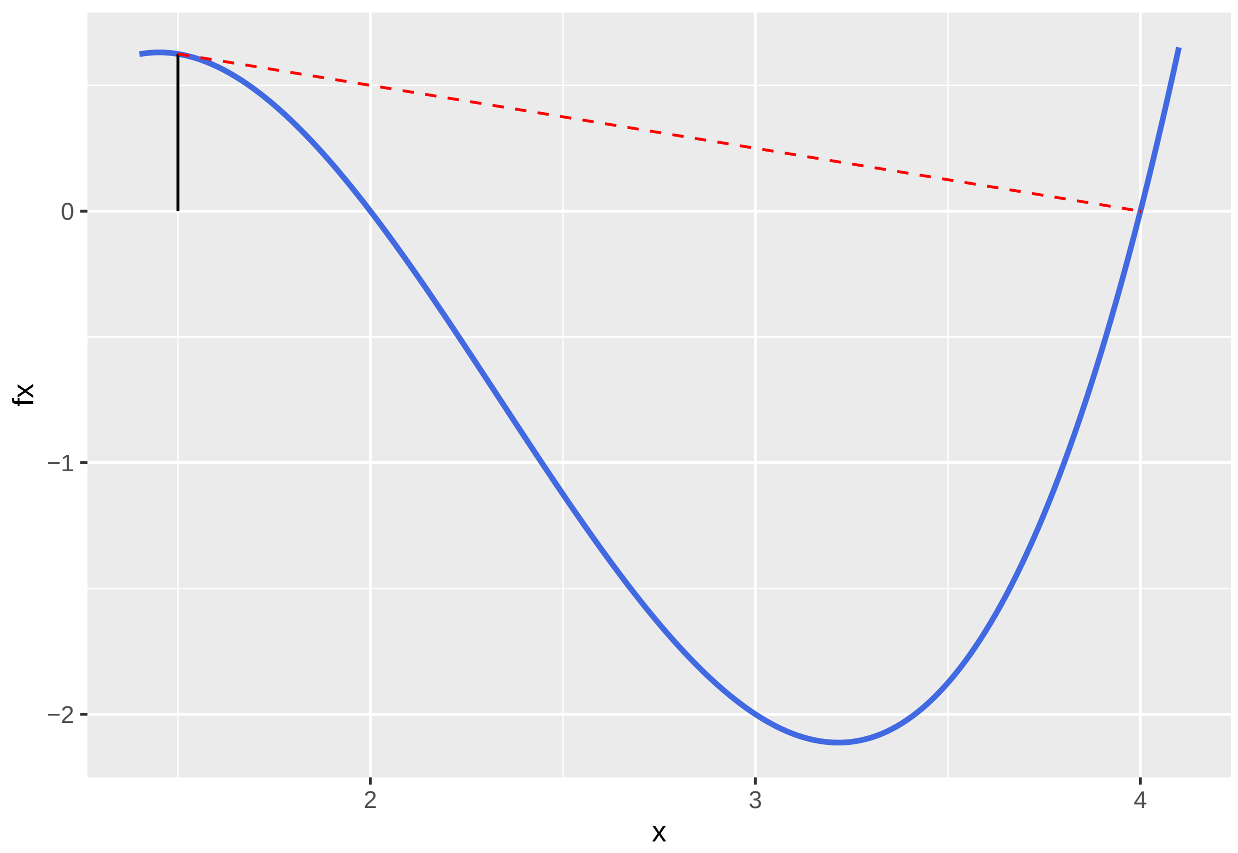
```
## [1] 1
```

Part (d) using x0 = 1.5

```
d3_1 <- function(x) {
  value <- x^3 - 7*x^2 + 14*x - 8
  derivative <- 3*x^2 - 14*x + 14
  return(c(value, derivative))
}

### x0 = 1.5
newtonraphson_show(d3_1, 1.5, iter = 8)
```

```
## Starting value is: 1.5
## Next x value: 4
## Next x value: 4
## Next x value: 4
## Next x value: 4
## Next x value: 4
## Next x value: 4
## Next x value: 4
## Next x value: 4
```



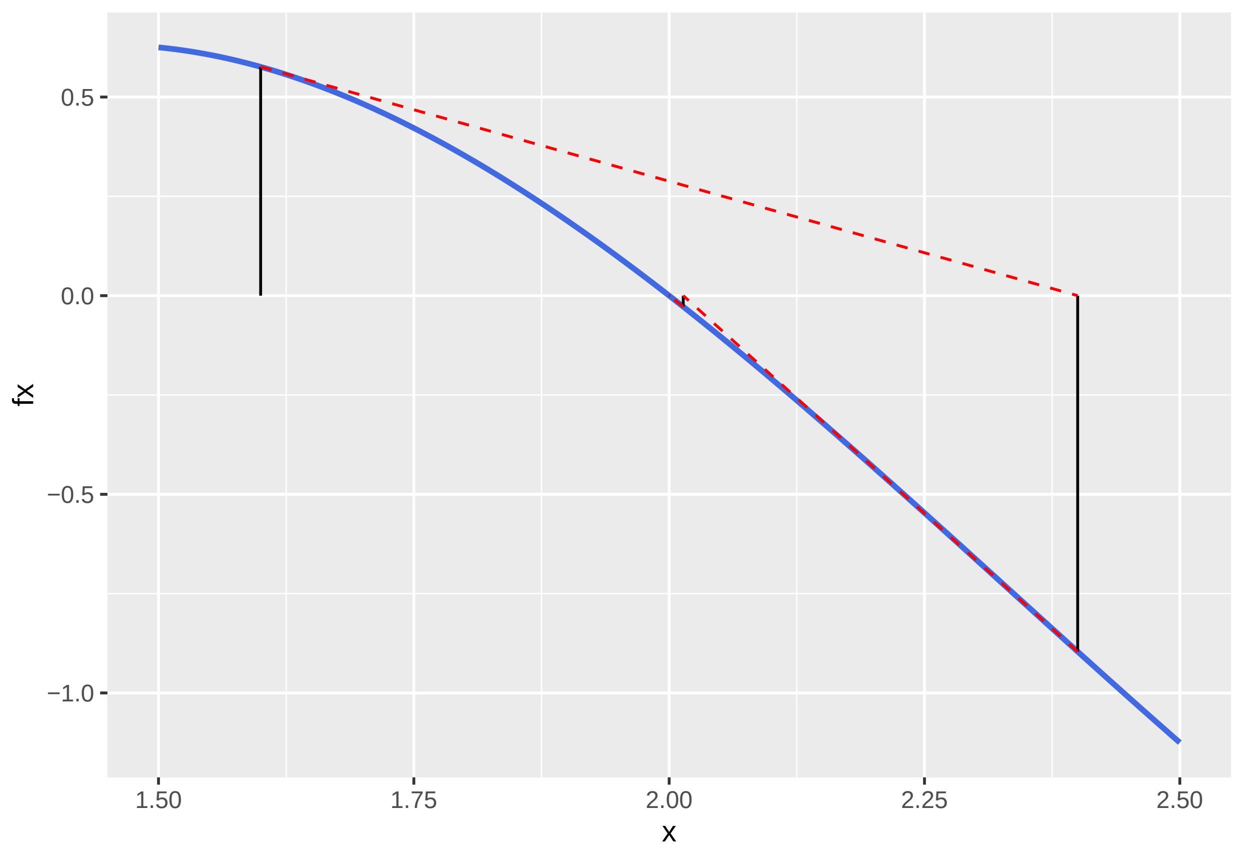
```
## [1] 4
```

Part (d) using x0 = 1.6

```
d3_1 <- function(x) {
  value <- x^3 - 7*x^2 + 14*x - 8
  derivative <- 3*x^2 - 14*x + 14
  return(c(value, derivative))
}

### x0 = 1.6
newtonraphson_show(d3_1, 1.6, iter = 8)
```

```
## Starting value is: 1.6
## Next x value: 2.4
## Next x value: 2.013793
## Next x value: 2.000091
## Next x value: 2
## Next x value: 2
## Next x value: 2
## Next x value: 2
## Next x value: 2
```



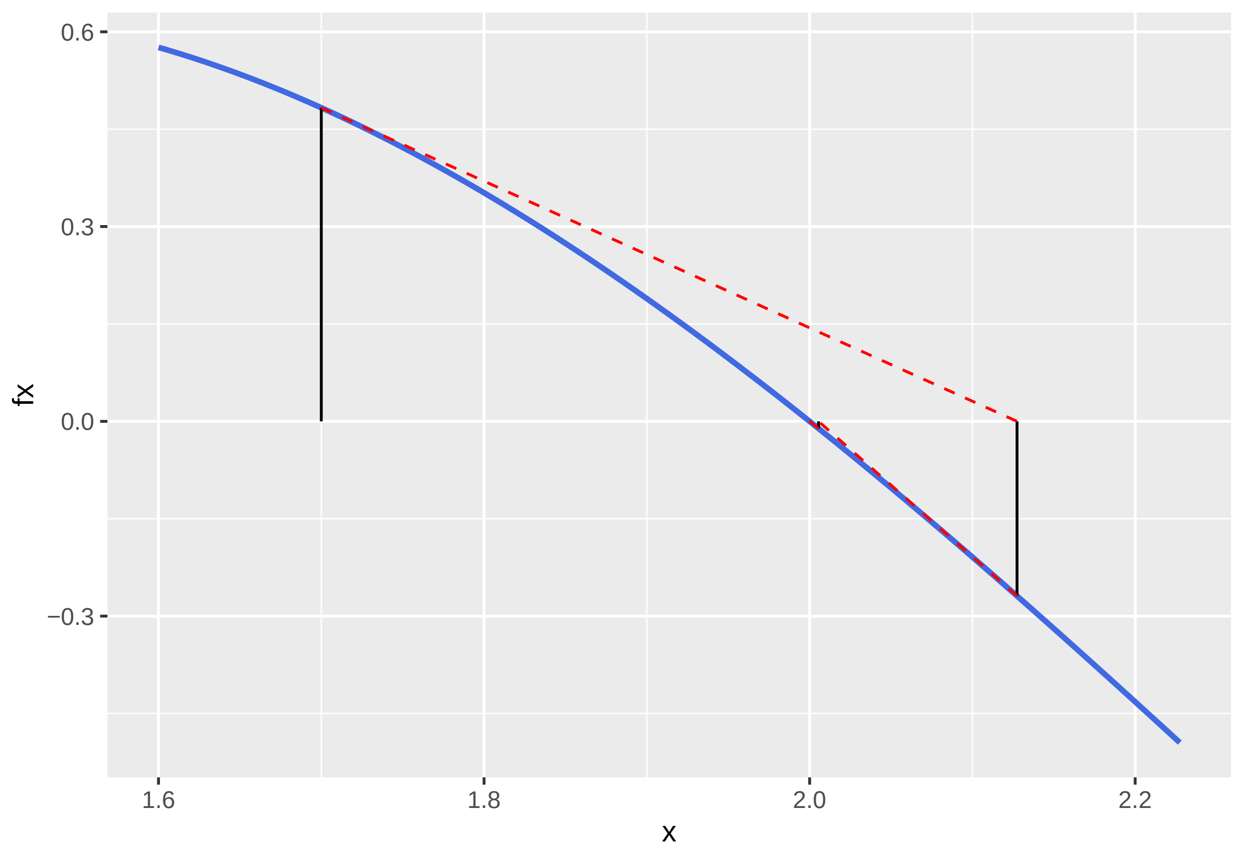
```
## [1] 2
```

Part (d) using x0 = 1.7

```
d3_1 <- function(x) {
  value <- x^3 - 7*x^2 + 14*x - 8
  derivative <- 3*x^2 - 14*x + 14
  return(c(value, derivative))
}

### x0 = 1.7
newtonraphson_show(d3_1, 1.7, iter = 8)
```

```
## Starting value is: 1.7
## Next x value: 2.127434
## Next x value: 2.005485
## Next x value: 2.000015
## Next x value: 2
## Next x value: 2
## Next x value: 2
## Next x value: 2
## Next x value: 2
```



```
## [1] 2
```

4. Root Finding with Secant Method [24 points- 20 points for completing the code. 1 pts each graph]

```
secant_show <- function(ftn, x0, x1, iter = 5) {
  df_points_1 <- data.frame(
    x1 = numeric(0),
    y1 = numeric(0),
    x2 = numeric(0),
    y2 = numeric(0))
  df_points_2 <- df_points_1
  cat("Starting values are:", x0, "and", x1, "\n")
  # the algorithm
  # very tricky and need to be smart about this algorithm
  for(i in 1:iter){
    x0old <- x0
    x1old <- x1
    f_xold_x0 <- ftn(x0)
    f_xold_x1 <- ftn(x1)
    xnew <- x1old - f_xold_x1 * ((x0old - x1old) / (f_xold_x0 - f_xold_x1))
    # update the point
    x1 <- xnew
    x0 <- x1old
    cat("Next x value:", xnew, "\n")
    # the line segments
    df_points_1[i, ] <- c(x0old, ftn(x0old), x1old, ftn(x1old))
    df_points_2[i, ] <- c(x1old, ftn(x1old), xnew, 0)
  }
  plot_start <- min(df_points_1$x0, df_points_1$x1, x0) - 0.1
  plot_end <- max(df_points_2$x0, df_points_2$x1, x0) + 0.1
  x <- seq(plot_start, plot_end, length.out = 200)
  fx <- rep(NA, length(x))
  for (i in seq_along(x)) {
    fx[i] <- ftn(x[i])
  }
  function_data <- data.frame(x, fx) # dataframe containing the function
  p <- ggplot(function_data, aes(x = x, y = fx)) +
    geom_line(color = "royalblue", size = 1) + # plot the function
    geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),
                 data = df_points_1, color = "black", lty = 1) +
    geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),
                 data = df_points_2, color = "red", lty = 2) +
    geom_abline(intercept = 0, slope = 1) # plot the line y = x
  print(p)
  x1
}
```

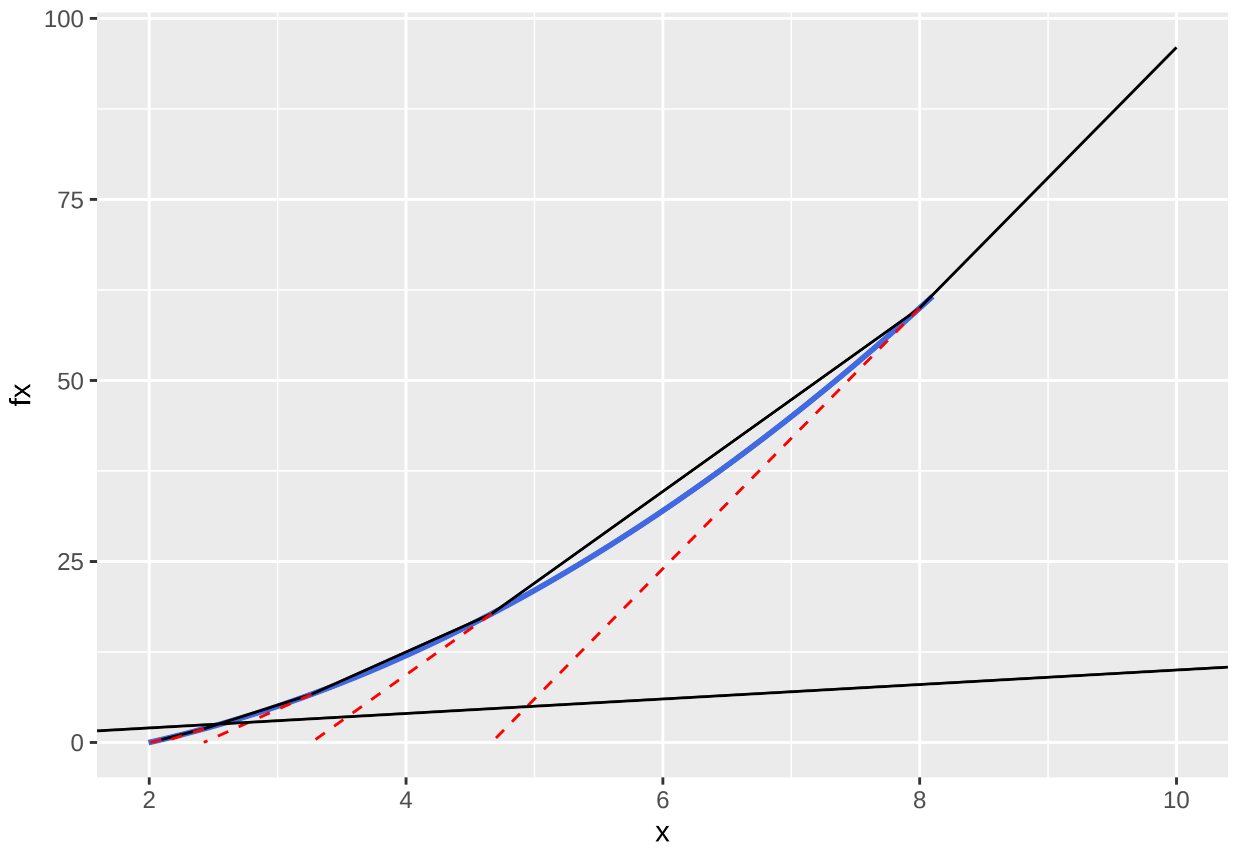
Produce graphs for:

The function $f(x) = x^2 - 4$ using x0 = 10, and x1 = 8

```
f4 <- function(x) x^2 - 4

### x0 = 10, x1 = 8
secant_show(f4, 10, 8, iter = 5)
```

```
## Starting values are: 10 and 8
## Next x value: 4.666667
## Next x value: 3.263158
## Next x value: 2.424779
## Next x value: 2.094333
## Next x value: 2.008867
```

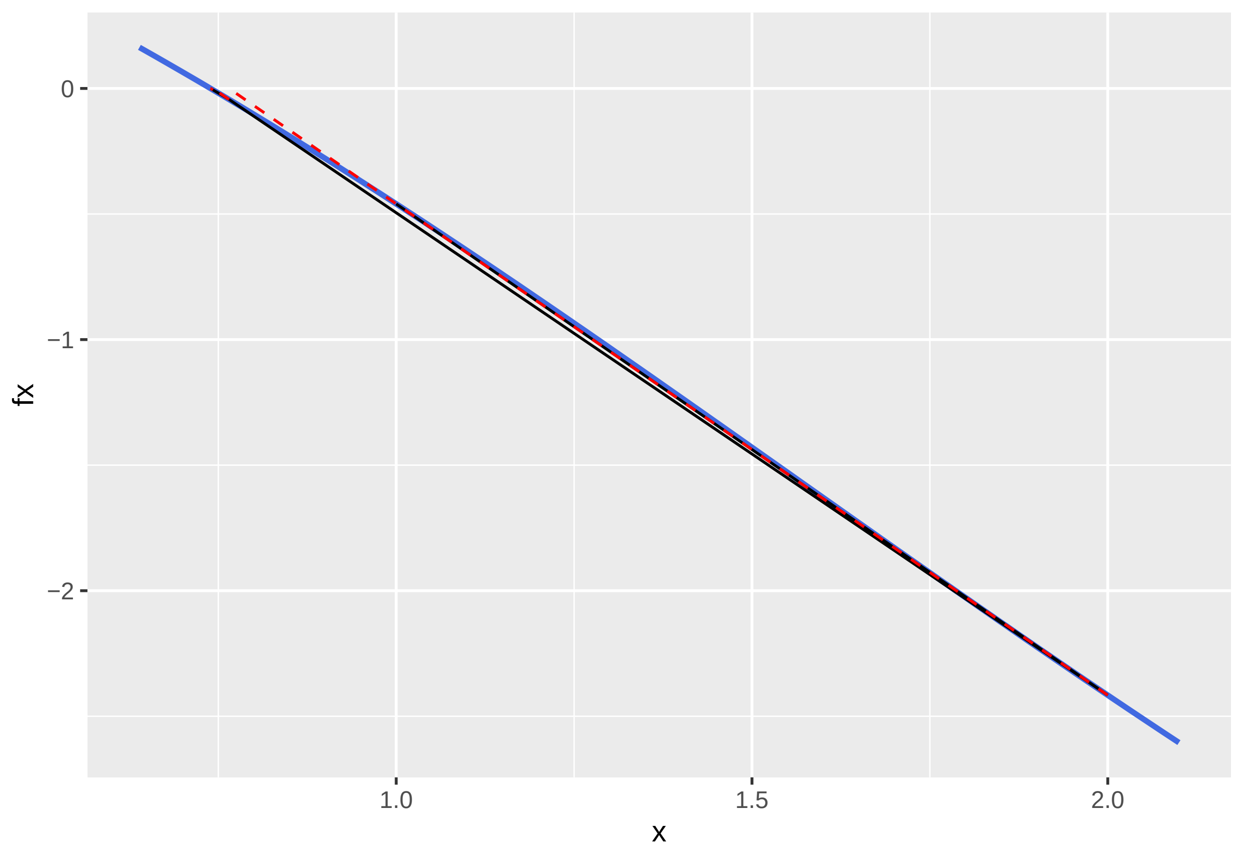


```
## [1] 2.008867
```

$f(x) = cos(x) - x$ using $x_0 = 1$ and $x_1 = 2$.

```
a4_1 <- function(x) cos(x) - x
### x0 = 1, x1 = 2
secant_show(a4_1, 1, 2, iter = 5)
```

```
## Starting values are: 1 and 2
## Next x value: 0.7650347
## Next x value: 0.7422994
## Next x value: 0.7391033
## Next x value: 0.7390851
## Next x value: 0.7390851
```

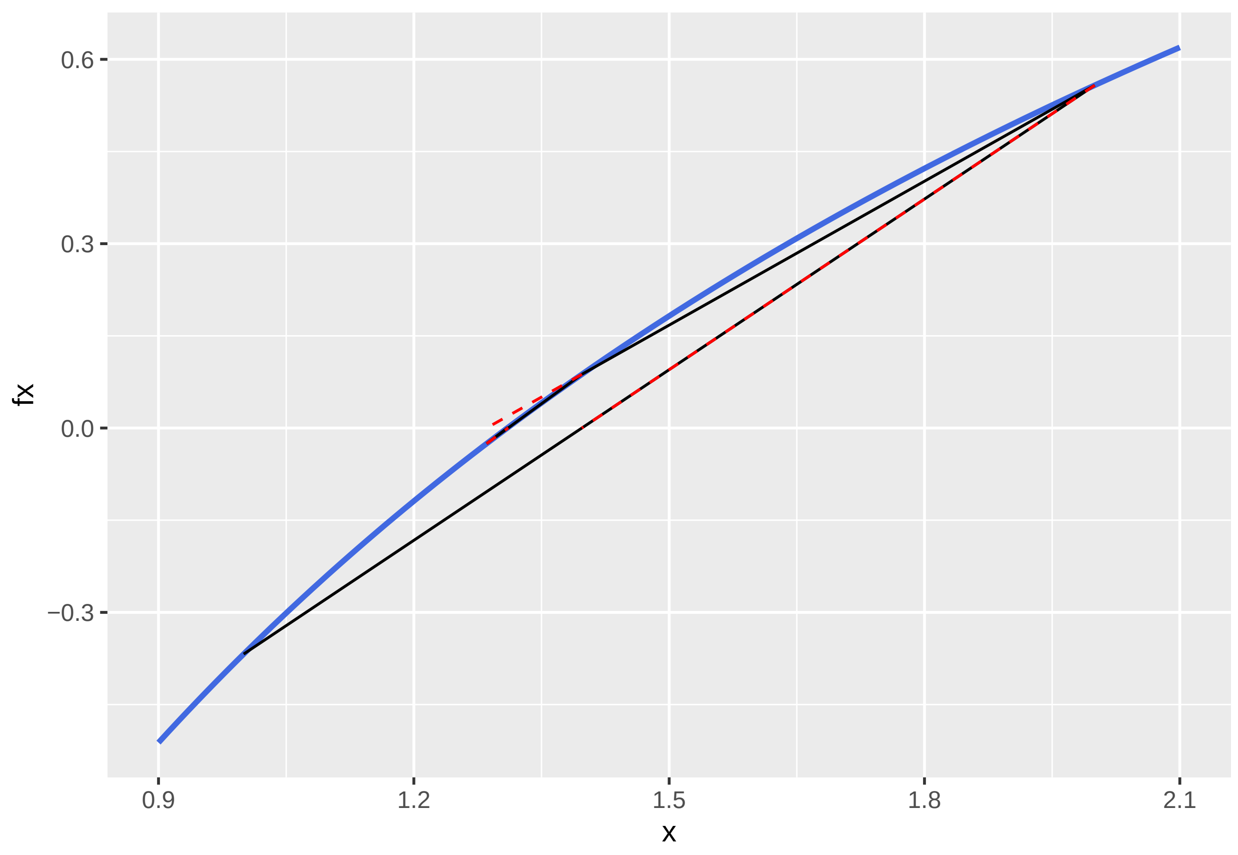


```
## [1] 0.7390851
```

$f(x) = log(x) - exp(-x)$ **using** $x_0 = 1$ **and** $x_1 = 2$**.**

```r
a4_2 <- function(x) log(x) - exp(-x)
### x0 = 1, x1 = 2
secant_show(a4_2, 1, 2, iter = 5)
```

```
## Starting values are: 1 and 2
## Next x value: 1.39741
## Next x value: 1.285476
## Next x value: 1.310677
## Next x value: 1.309808
## Next x value: 1.3098
```

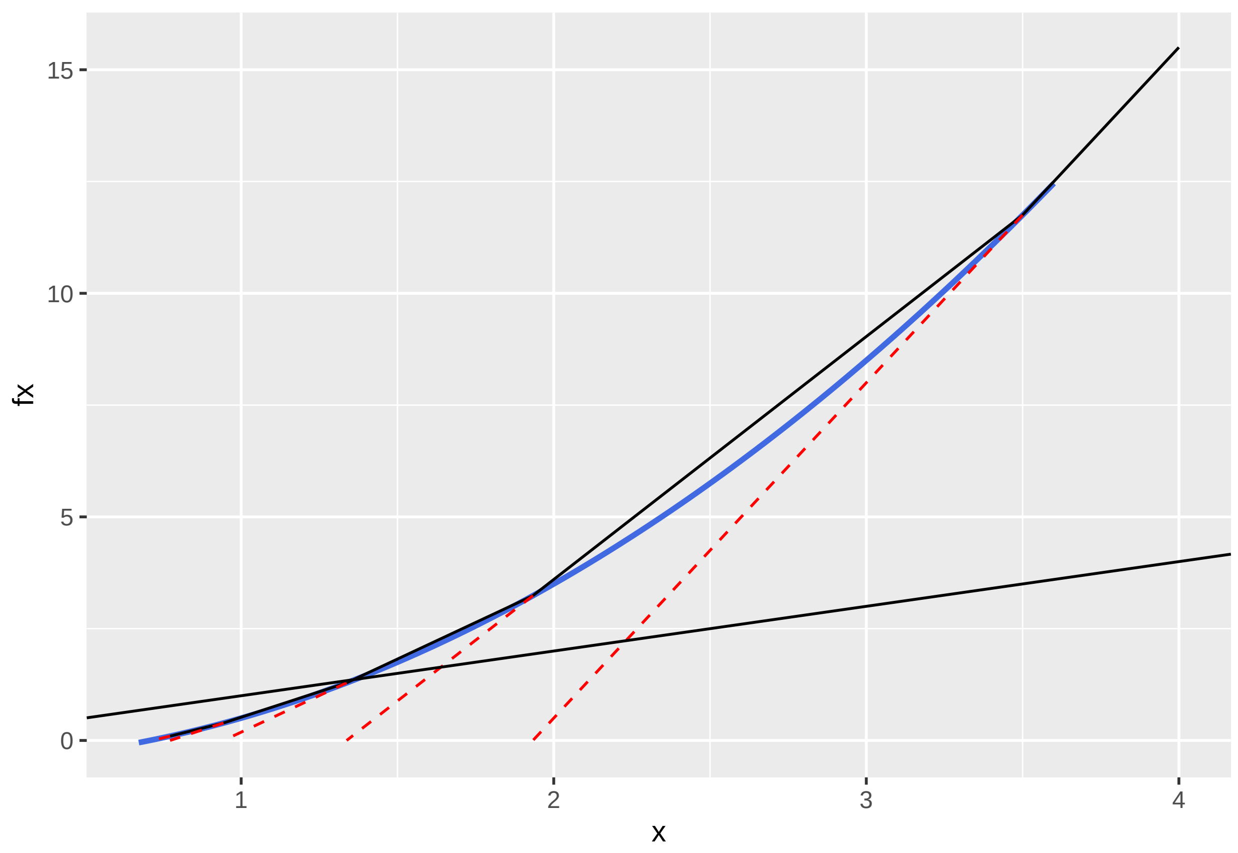


```
## [1] 1.3098
```

Find the root of $x^2 - 0.5$ using $x_0 = 4$ and $x_1 = 3.5$.

```
a4_3 <- function(x) x^2 - 0.5
### x0 = 4, x1 = 3.5
secant_show(a4_3, 4, 3.5, iter = 5)
```

```
## Starting values are: 4 and 3.5
## Next x value: 1.933333
## Next x value: 1.337423
## Next x value: 0.9434163
## Next x value: 0.7724116
## Next x value: 0.7161008
```



```
## [1] 0.7161008
```

5. Coordinate Descent Algorithm for Optimization [20 points]

```
##### A modifcation of code provided by Eric Cai
golden = function(f, lower, upper, tolerance = 1e-5)
{
   golden.ratio = 2/(sqrt(5) + 1)

   ## Use the golden ratio to find the initial test points
   x1 <- lower + golden.ratio * (upper - lower)
   x2 <- upper - golden.ratio * (upper - lower)

   ## the arrangement of points is:
   ## lower ----- x2 --- x1 ----- upper

   ### Evaluate the function at the test points
   f1 <- f(x1)
   f2 <- f(x2)

   while (abs(upper - lower) > tolerance) {
        if (f2 > f1) {
        # the minimum is to the right of x2
        lower <- x2  # x2 becomes the new lower bound
        x2 <- x1     # x1 becomes the new x2
        f2 <- f1     # f(x1) now becomes f(x2)
        x1 <- lower + golden.ratio * (upper - lower)
        f1 <- f(x1)  # calculate new x1 and f(x1)
        } else {
        # then the minimum is to the left of x1
        upper <- x1  # x1 becomes the new upper bound
        x1 <- x2     # x2 becomes the new x1
        f1 <- f2
        x2 <- upper - golden.ratio * (upper - lower)
        f2 <- f(x2)  # calculate new x2 and f(x2)
        }
    }
    (lower + upper)/2 # the returned value is the midpoint of the bounds
}
```

Example for Problem 5

```
f5 <- function(x) { (x - 3)^2 }

### lower = 0, upper = 10, tolerance = 1e-5
golden(f5, 0, 10)
```

```
## [1] 3.000001
```

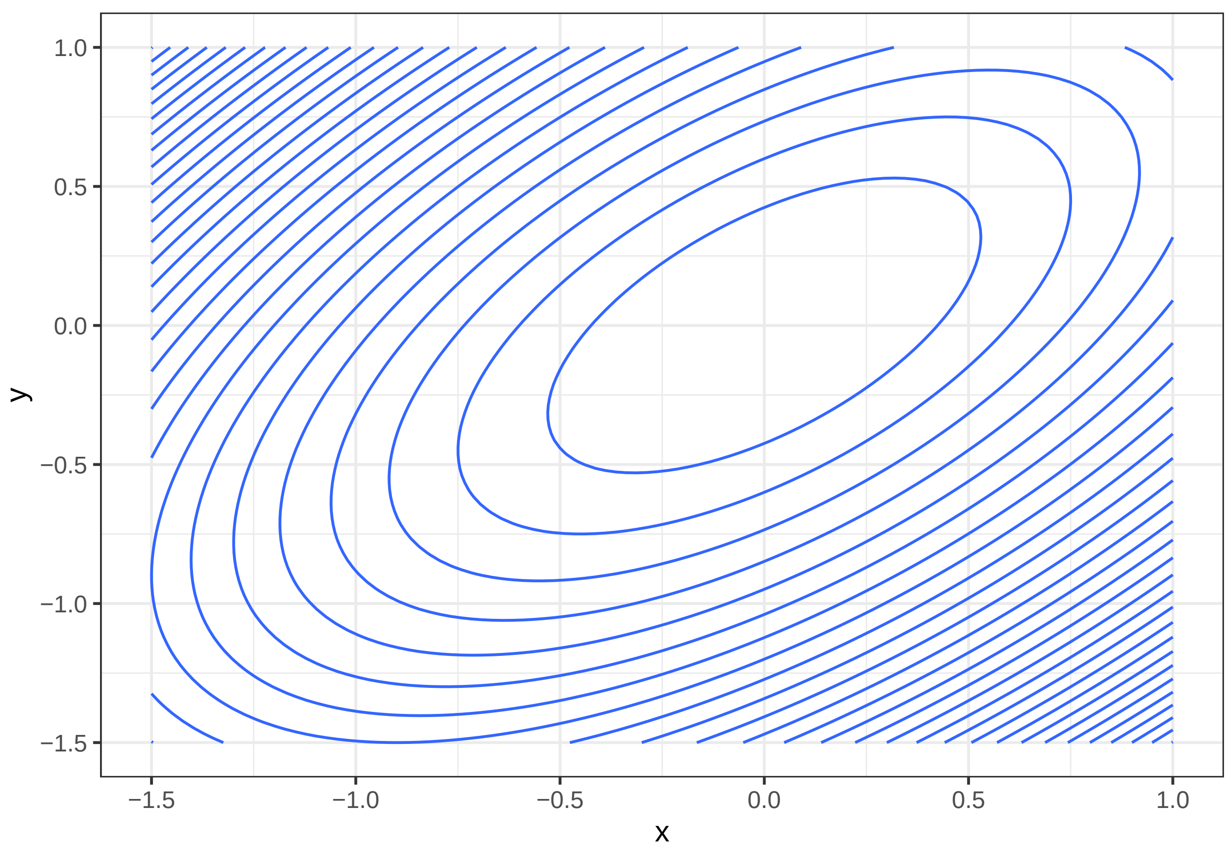
Creating a new function for Problem 5:

```
### I created a function for the 15 iterations for various starting points
contour_mapping <- function(a, b) {
  g <- function(x, y) {
    5 * x ^ 2 - 6 * x * y + 5 * y ^ 2
  }
  x <- seq(-1.5, 1, len = 100)
  y <- seq(-1.5, 1, len = 100)
  x_i <- a
  y_i <- b
  x0 <- 0
  y0 <- 0
  cat("Starting values are:", x_i, "and", y_i, "\n")
  for(i in 1:15) {
    x0 <- x_i
    y0 <- y_i
    g_x <- function(x) {
      # x = x, y = y0
      g_x_y0 <- g(x, y0)
      return(g_x_y0)
    }
    x0 <- golden(g_x, lower = -1.5, upper = 1.5, tolerance = 1e-5)
    g_y <- function(y) {
      # x = x0, y = y
      g_x0_y <- g(x0, y)
      return(g_x0_y)
    }
    y0 <- golden(g_y, lower = -1.5, upper = 1.5, tolerance = 1e-5)
    x_i <- x0
    y_i <- y0
    cat("Next values of x & y are:", x0, "&", y0, "\n")
    # the line segment
    # I am currently working on figuring out my line segment for my new function
  }
  contour_df <- data.frame(
    x = rep(x, each = 100),
    y = rep(y, 100),
    z = outer(x, y, g)[1:100^2]
  )
  p <- ggplot(contour_df, aes(x = x, y = y, z = z)) +
    geom_contour(binwidth = 0.9) +
    theme_bw()
  print(p)
}
```

Graph for starting point x = -1.5, and y = -1.5.

```
### a = -1.5, b = -1.5
contour_mapping(-1.5, -1.5)
```

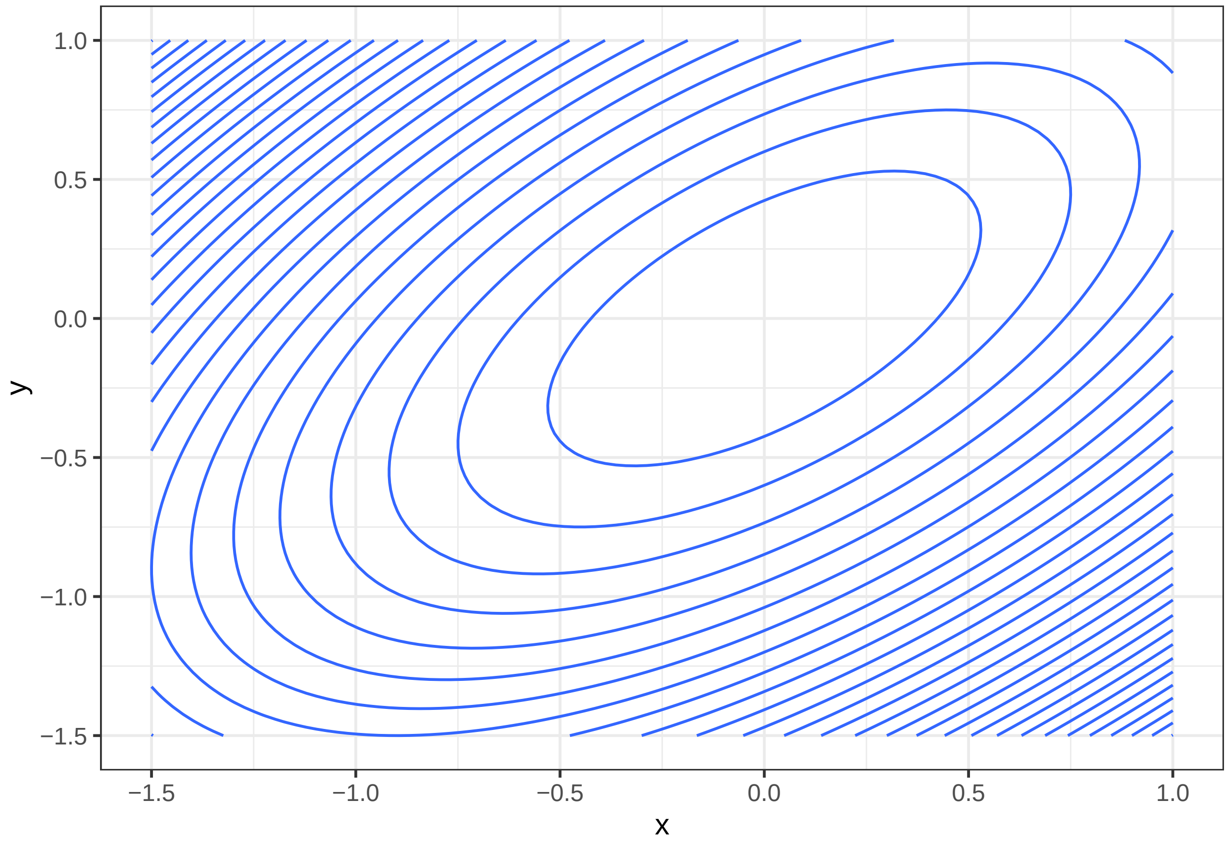
```
## Starting values are: -1.5 and -1.5
## Next values of x & y are: -0.9000005 & -0.5399991
## Next values of x & y are: -0.3239996 & -0.1943994
## Next values of x & y are: -0.1166404 & -0.06998455
## Next values of x & y are: -0.04199138 & -0.02519619
## Next values of x & y are: -0.01511759 & -0.009071722
## Next values of x & y are: -0.005444077 & -0.003266769
## Next values of x & y are: -0.001961105 & -0.001174652
## Next values of x & y are: -0.0007038706 & -0.0004213549
## Next values of x & y are: -0.0002519686 & -0.0001515037
## Next values of x & y are: -8.941291e-05 & -5.364775e-05
## Next values of x & y are: -3.315613e-05 & -1.788258e-05
## Next values of x & y are: -1.105204e-05 & -6.830539e-06
## Next values of x & y are: -4.221505e-06 & -4.221505e-06
## Next values of x & y are: -4.221505e-06 & -4.221505e-06
## Next values of x & y are: -4.221505e-06 & -4.221505e-06
```



Graph for starting point x = -1.5, and y = 1.

```
### a = -1.5, b = 1
contour_mapping(-1.5, 1)
```

```
## Starting values are: -1.5 and 1
## Next values of x & y are: 0.6000003 & 0.3600014
## Next values of x & y are: 0.2160011 & 0.1296002
## Next values of x & y are: 0.07775905 & 0.04665583
## Next values of x & y are: 0.02799318 & 0.01679518
## Next values of x & y are: 0.01007699 & 0.00604587
## Next values of x & y are: 0.003627645 & 0.002177308
## Next values of x & y are: 0.001305664 & 0.0007822315
## Next values of x & y are: 0.0004681721 & 0.0002809033
## Next values of x & y are: 0.0001693863 & 0.000100465
## Next values of x & y are: 6.209076e-05 & 3.576517e-05
## Next values of x & y are: 2.210409e-05 & 1.366108e-05
## Next values of x & y are: 6.830539e-06 & 4.221505e-06
## Next values of x & y are: 4.221505e-06 & 4.221505e-06
## Next values of x & y are: 4.221505e-06 & 4.221505e-06
## Next values of x & y are: 4.221505e-06 & 4.221505e-06
```



6. Extra Credit: Bisection Search Graph [up to 10 points]

I am unable to do it since I have a midterm on Friday (2/28/2020). Am I able to have an extension on this extra credit? I feel as if I can complete this.