Stats 102A - Homework 5

Put Your Name Here

Homework 3 Requirements

You will submit two files.

The files you submit will be:

1. `102a_hw_05_output_First_Last.Rmd` Take the provided R Markdown file and make the necessary edits so that it generates the requested output.
2. `102a_hw_05_output_First_Last.pdf` Your output PDF file. This is the primary file that will be graded. Make sure all requested output is visible in the output file.

There is no script file to submit

Failure to submit all files will result in an automatic 40 point penalty.

Academic Integrity

At the top of your R markdown file, be sure to include the following statement after modifying it with your name.

"By including this statement, I, Joe Bruin, 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."

If you collaborated verbally with other students, please also include the following line to credit them.

"I did discuss ideas related to the homework with Josephine Bruin for parts 2 and 3, with John Wooden for part 2, and with Gene Block for part 5. At no point did I show another student my code, nor did I look at another student's code."

Reading

- SPURS: https://www.taylorfrancis.com/books/9780429143335
  - Chapter 9
  - Chapter 10
  - Chapter 12 sections 1-2

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?

b. How would the value 5.5 be stored in this system?
c. What value would the following bit sequence represent `0 0111 00000`?
d. What value would the following bit sequence represent `0 0111 00001`? (Express as a fraction and also in decimal.)
e. What is the smallest positive normalized value that can be expressed? (Fill in the bits. Express as a fraction and also in decimal.)
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.)
g. What is the largest denormalized value that can be expressed? (Fill in the bits. Express as a fraction and also in decimal.)
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.)
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?
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)
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)

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

This is a modified version of SPURS chapter 10, section 6, exercise 4.

I have written my own version of fixedpoint_show, which uses ggplot to produce the graphs. The code performs fixed point iteration to find a solution to $f(x) = x$.

I have modified it so that it works better for R Markdown output. Instead of prompting the user to continue, it will perform a number of iterations as specified by the `iter` value in the parameters. I encourage you to read through the code line by line and make sure you understand it.

- Do part (a) using x0 = 1
- Do part (a) using x0 = 3
- Do part (a) using x0 = 6
- Do part (b) using x0 = 2
- Do part (c) using x0 = 2
- Do part (d) using x0 = 2, no more than 6 iterations

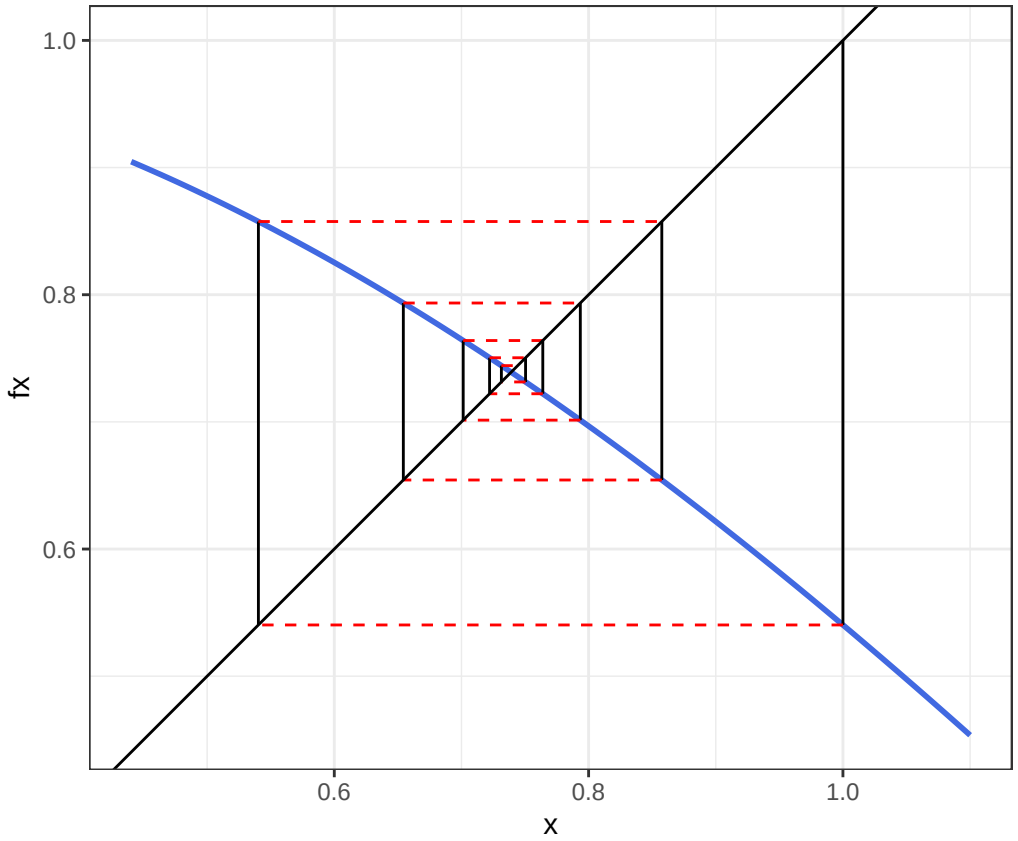
note that parts b, c, d are all ways of rewriting f(x) = log(x) - exp(-x) = 0 as g(x) = x

```
library(ggplot2)
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
```

```r
  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
}

## Part a, x0 = 1
f <- function(x) cos(x)
fixedpoint_show(f, 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
```

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

Modified version of SPURS chapter 10, section 6, exercise 5.

For this problem, we are implementing the Newton Raphson method for root finding.

I've written some of the code for you. You'll need to write the rest of the code so it can use ggplot to show the function and lines for each iteration.

You'll probably want to refer to the code for the fixed point iteration. Also pay attention to the example used in the textbook for how the function should be programmed. The function takes a value of x and returns two values: the value of the function, and the value of the derivative at that point. Refer to section 10.3, especially page 176.

Once you have it running, produce graphs for:

- The function f(x) = x^2 - 4 using x0 = 10
- part (a) using x0 = 1, 3, 6 Results should be similar to finding fixed point of cos(x)
- part (b) using x0 = 2 Results should be similar to finding fixed point of exp(exp(-x))
- part (c) using x0 = 0
- 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)

```r
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(NA, NA, NA, NA) # vertical segment
    df_points_2[i,] <- c(NA, NA, NA, NA) # tangent segment
  }


  p <- ggplot( ... ) + ... # write your code

  print(p)

  xnew # value that gets returned
}


## Part a
# example of how your functions could be written
a <- function(x){
  value <- cos(x) - x   # f(x)
  derivative <- -sin(x) - 1  # f'(x)
  c(value, derivative) # the function returns a vector with two values
}

newtonraphson_show(a, 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
```

```
## Error in ggplot(...): '...' used in an incorrect context
```

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

Modified version of SPURS chapter 10, section 6, exercise 6.

Implement the secant method for root finding. Write a function called `secant_show` similar to the `newtonraphson_show` and `fixedpoint_show` functions. In your function, perform iterations of the algorithm and plot the results. It should behave in a very similar fashion to `newtonraphson_show`.

A framework of the function has been provided. It will take 4 inputs:

- The function. It takes a value of x and returns the value of f(x). (no need to return the derivative)
- x0 The x-value at iteration 0
- x1 The x-value at iteration 1. The secant uses the last two values, and draws a secant line that connects those until that secant line intersects the x-axis.
- the number of iterations to perform

While non-trivial, I do believe that the previous examples will provide a good guide for writing your code.

When drawing the secant line, keep in mind that what will be your start and end point can be different depending on where xnew will be. (For example, xnew can be to the left of x0 and x1, it can be between x0 and x1, and it can be to the right of x0 and x1.) In each case, the starting point of the line segment should be the left-most point, and the end-point of the line segment should be the right-most point. **You cannot assume that xnew will always be one of the endpoints.**

- The function $f(x) = x^2 - 4$ using x0 = 10, and x1 = 8
- Use your function to find the roots of $cos(x) - x$ using $x_0 = 1$ and $x_1 = 2$.
- Use your function to find the root of $log(x) - exp(-x)$ using $x_0 = 1$ and $x_1 = 2$.
- Find the root of $x^2 - 0.5$ using $x_0 = 4$ and $x_1 = 3.5$.

```
secant_show <- function(ftn, x0, x1, iter = 5) {


}
```

5. Coordinate Descent Algorithm for Optimization [20 points]

Coordinate descent is an optimization algorithm. The algorithm attempts to find a local minimum of a function. We perform a search in one direction to find the value that minimizes the function in that direction while the other values are held constant. Once the value for that direction is updated, you perform the same operation for the other coordinate directions. This repeats until it has been updated for all coordinate directions, at which point the cycle repeats.

Thus for a function of two variables $f(x, y)$, a simple version of the algorithm can be described as follows:

1) Start with some initial values of x and y. This is time 0, so we have $x^{(0)}$ and $y^{(0)}$.
2) Iterate:
    1) Update $x^{(t+1)}$ to be the value of x that minimizes $f(x, y = y^{(t)})$
    2) Update $y^{(t+1)}$ to be the value of y that minimizes $f(x = x^{(t+1)}, y)$
3) Stop when some convergence criterion has been met.

The "tricky" part of the algorithm is finding the value that minimizes the function along one of the directions.

Golden Section Search Method (with Video)

This unidimensional minimization be done in one of many ways, but for our purposes, we will use the golden section search method.

The premise of how the golden section search works is summarized very nicely in this video from CUBoulder-Computing: https://vimeo.com/86277921

I will provide the code for the golden section search method here. This is a modified version of Eric Cai's code (https://chemicalstatistician.wordpress.com). It has been modified so that the locations of x1 and x2 match the CUBoulderComputing video

```
##### 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
}
```

We can thus use the golden search to find the minimizing value of a function. For example, the function $f(x) = (x-3)^2$ has a minimum value at $x = 3$.

```
f <- function(x){ (x - 3)^2 }
golden(f, 0, 10)
```

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

Back to Coordinate Descent

With our golden search function, we can now create our coordinate descent algorithm:

1) Start with some initial values of x and y. This is time 0, so we have $x^{(0)}$ and $y^{(0)}$.
2) Iterate:
    1) Update x:
        a. Find the function $f(x) = f(x, y = y^{(t)})$
        b. Use golden search to minimize $f(x)$
        c. Set $x^{(t+1)}$ be the result of the search.
    2) Update y
        a. Find the function $f(y) = f(x = x^{(t+1)}, y)$
        b. Use golden search to minimize $f(y)$
        c. Set $y^{(t+1)}$ be the result of the search.
3) Stop when some convergence criterion has been met.

Write code to perform coordinate descent to minimize the following function:

$$g(x, y) = 5x^2 - 6xy + 5y^2$$

```
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)
```

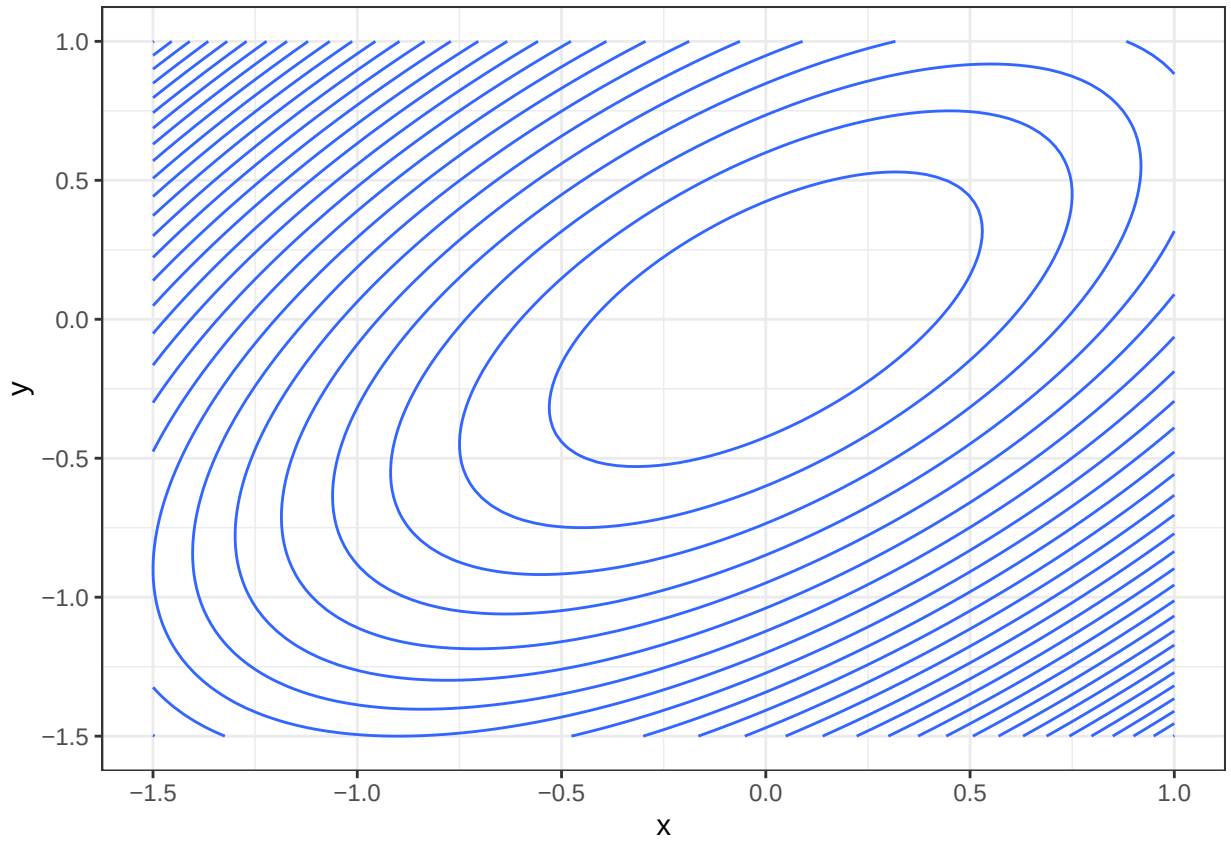
Requirements for this task:

1) Your search space for the golden search can be limited to the range -1.5 to 1.5 for both the x and y directions.
2) For your starting point, use x = -1.5, and y = -1.5.
3) For the first step, hold y constant, and find the minimum in the x direction.
4) Plot the segments showing each 'step' of the algorithm onto the contour plot.
5) After each full iteration, print out the current values of x and y. Hint: after your first full iteration, the next location should be (-0.9, -0.54).
6) Stop after 15 full iterations, or if the difference between one x and the next is less then `1e-5`. The true minimum is at (0,0). Your code should come close to that.

My complete solution is 12 lines. Of course yours might be longer, but that should give you an idea of how complicated it needs to be.

7) Once you have completed the above, do it again, but with a starting location of x = -1.5, and y = 1. Again, for the first step, hold y constant. You are allowed to copy and paste your working code and just alter the starting location. You do not need to create a brand new function.

```
contour_df <- data.frame(
  x = rep(x, each = 100),
  y = rep(y, 100),
  z = outer(x, y, g)[1:100^2]
  )
ggplot(contour_df, aes(x = x, y = y, z = z)) +
  geom_contour(binwidth = 0.9) +
  theme_bw()
```




```
# write your code here
x_i <- -1.5
y_i <- -1.5
```

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

Write a function called `bisection_show` which will find a root using the bisection method.

It should produce a graph that shades the portions that are eliminated, similar to the picture I drew in this video at 41:21.

https://youtu.be/NttyKvokva8?t=2481

Have it search for a root for $f(x) = x^2 - 9$ on the interval 0 to 10. Show the progression by producing a plot after 1 iteration, 2 iterations, 3 iterations, and 4 iterations.