27/40

let me know if the comments make sense
(-5 for 1 day late submission)

Bios 6301: Assignment 5

Jonathan Lifferth

Due Thursday, 12 October, 1:00 PM

$5^{n=day}$ points taken off for each day late.

40 points total.

Submit a single knitr file (named `homework5.rmd`), along with a valid PDF output file. Inside the file, clearly indicate which parts of your responses go with which problems (you may use the original homework document as a template). Add your name as `author` to the file's metadata section. Raw R code/output or word processor files are not acceptable.

Failure to name file `homework5.rmd` or include author name may result in 5 points taken off.

Question 1

15 points

A problem with the Newton-Raphson algorithm is that it needs the derivative f'. If the derivative is hard to compute or does not exist, then we can use the *secant method*, which only requires that the function f is continuous.

Like the Newton-Raphson method, the **secant method** is based on a linear approximation to the function f. Suppose that f has a root at a. For this method we assume that we have *two* current guesses, x_0 and x_1, for the value of a. We will think of x_0 as an older guess and we want to replace the pair x_0, x_1 by the pair x_1, x_2, where x_2 is a new guess.

To find a good new guess x2 we first draw the straight line from $(x_0, f(x_0))$ to $(x_1, f(x_1))$, which is called a secant of the curve $y = f(x)$. Like the tangent, the secant is a linear approximation of the behavior of $y = f(x)$, in the region of the points x_0 and x_1. As the new guess we will use the x-coordinate x_2 of the point at which the secant crosses the x-axis.

The general form of the recurrence equation for the secant method is:

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

Notice that we no longer need to know f' but in return we have to provide *two* initial points, x_0 and x_1.

Write a function that implements the secant algorithm. Validate your program by finding the root of the function $f(x) = \cos(x) - x$. Compare its performance with the Newton-Raphson method – which is faster, and by how much? For this example $f'(x) = -\sin(x) - 1$.

```
secant <- function(f, x0, x1, tol = 1e-6, max_iter = 100) {

# f: A function
# x0: initial guess
# x1: The second guess
# tol: error tolerance
```

```
# max_iter: The maximum number of iterations.

    for (i in 1:max_iter) {
    fx0 <- f(x0)
    fx1 <- f(x1)

    # Calculate next approximation for root.
    x2 <- x1 - fx1 * (x1 - x0) / (fx1 - fx0)

    # Check if the convergence criterion is met.
    if (abs(x2 - x1) < tol) {
      return(x2)
    }

    # Update guesses for root.
    x0 <- x1
    x1 <- x2
  }

  # root could not be found.
  return(NULL)
}
```

```
# here I test my secant function on another simple function
f <- function(x) x^2 - 1
x0 <- 5
x1 <- 10
root <- secant(f, x0, x1)
root
```

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

```
# here, I find the root of the function $f(x) = \cos(x) - x$
secant(function(x) cos(x) - x, x0, x1)
```

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

```
# here's the Newton-Raphson algorithm

x<-10
while(abs(f <- x^3 + x^2 - 3*x -3) > 1e-8) {
  fp <- 3*x^2 + 2*x - 3
  x <- x - f / fp
}
x
```

i think this arrow may be the source of your issue

```
## [1] 1.732051
```

```
# Compare its performance with the Newton-Raphson method -- which is faster, and by how much?

x0 <- 5
x1 <- 10
```

```r
# secant
secant(function(x) cos(x) - x, x0, x1)
```

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

```r
# Newton-Raphson
#x3 <- 10
#while(abs(f <- cos(x3) - x3) > 1e-8) {
#  fp <- sin(x3) - 1
#  x3 <- x3 - f / fp
#}
#x3
```

I don't know why I'm getting the following error when I try to run the Newton-Raphson method :

Line 93 "missing value where TRUE/FALSE needed"

Question 2

20 points

The game of craps is played as follows (this is simplified). First, you roll two six-sided dice; let x be the sum of the dice on the first roll. If x = 7 or 11 you win, otherwise you keep rolling until either you get x again, in which case you also win, or until you get a 7 or 11, in which case you lose.

Write a program to simulate a game of craps. You can use the following snippet of code to simulate the roll of two (fair) dice:

```r
play_craps <- function(seed, show_progress=TRUE) {
  set.seed(seed)
  game_over <- FALSE

  x <- sum(ceiling(6*runif(2)))
  if (show_progress) {
    print(c("x:", x))
  }
  if(x == 7 | x == 11) {
    print("You win!")
    game_over == TRUE
  } else {
    while(game_over == FALSE) {
      y <- sum(ceiling(6*runif(2)))
      if (show_progress) {
        print(c("y: ", y))
      }
      if (y == x) {
        print("You win!")
        game_over <- TRUE
      } else if (y == 7 | y == 11) {
        print("You lose :(")
```

```
        game_over <- TRUE
        }
    }
  }

  print("Game over ")
}


play_craps(seed=100, show_progress = FALSE)
```

```
## [1] "You lose :("
## [1] "Game over "
```

1. The instructor should be able to easily import and run your program (function), and obtain output that clearly shows how the game progressed. Set the RNG seed with `set.seed(100)` and show the output of three games. (lucky 13 points)

```
play_craps(100)
```

```
## [1] "x:" "4"
## [1] "y: " "5"
## [1] "y: " "6"
## [1] "y: " "8"
## [1] "y: " "6"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "y: " "8"
## [1] "y: " "9"
## [1] "y: " "9"
## [1] "y: " "5"
## [1] "y: " "11"
## [1] "You lose :("
## [1] "Game over "
```

```
play_craps(100)
```

```
## [1] "x:" "4"
## [1] "y: " "5"
## [1] "y: " "6"
## [1] "y: " "8"
## [1] "y: " "6"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "y: " "8"
## [1] "y: " "9"
## [1] "y: " "9"
```

```
## [1] "y: " "5"
## [1] "y: " "11"
## [1] "You lose :("
## [1] "Game over "
```

```
play_craps(100)
```

```
## [1] "x:" "4"
## [1] "y: " "5"
## [1] "y: " "6"
## [1] "y: " "8"
## [1] "y: " "6"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "y: " "8"
## [1] "y: " "9"
## [1] "y: " "9"
## [1] "y: " "5"
## [1] "y: " "11"
## [1] "You lose :("
## [1] "Game over "
```

with your method, you are running 1 game three times, you should set the seed outside of the function so that you are running 3 games one time (does that make sense?)

1. Find a seed that will win ten straight games. Consider adding an argument to your function that disables output. Show the output of the ten games. (7 points)

```
for(i in 1:10){
  play_craps(1)
}
```

```
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
```

```
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
## [1] "x:" "5"
## [1] "y: " "10"
## [1] "y: " "8"
## [1] "y: " "10"
## [1] "y: " "5"
## [1] "You win!"
## [1] "Game over "
```

I heard after the fact that Cole wanted us to set the seed outside of the function but the instructions don't technically say that?

Question 3 yes, but with your method you are doing the same game everytime so you found a seed that wins one game. not ten.

5 points

This code makes a list of all functions in the base package:

```
objs <- mget(ls("package:base"), inherits = TRUE)
funs <- Filter(is.function, objs)
```

Using this list, write code to answer these questions.

1. Which function has the most arguments? (3 points)

```
objs <- mget(ls("package:base"), inherits = TRUE)
funs <- Filter(is.function, objs)

top_fun <- ""
top_fun_len <- 0

for(i in 1:length(funs)) {
  fun_len <- length(formals(names(funs[i])))
  if (fun_len > top_fun_len) {
    top_fun <- names(funs[i])
    top_fun_len <- length(formals(names(funs[i])))
  }
}

print(top_fun_len)
```

```
## [1] 22
```

```
print(top_fun)
```

```
## [1] "scan"
```

1. How many functions have no arguments? (2 points)

```
no_args_count <- 0
for(i in 1:length(funs)) {
  fun_len <- length(formals(names(funs[i])))
  if (fun_len == 0) {
    no_args_count <- no_args_count + 1
  }
}
no_args_count
```

```
## [1] 227
```

Hint: find a function that returns the arguments for a given function.