Bios 6301: Assignment 4

Hannah Weeks

Due Tuesday, 27 October, 1:00 PM

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

50 points total.

Submit a single knitr file (named `homework4.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 `homework4.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 x_2 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 $f'(x) = -\sin(x) - 1$.

```r
#Functions to be used to test methods
fTest <- function(x) cos(x) - x
fDerivTest <- function(x) -sin(x) - 1

#Approximate actual solution
#0.739085
```

First, write a function for the secant method:

```r
#fRoot = function we want to find the root of
#x.old, x.new = initial guesses
#max.iter = in case of divergence or initial guesses not close enough
secant <- function(fRoot, x.old, x.new, tol = 1e-10, max.iter = 1000, print = TRUE){
  iter <- 1
  test <- fRoot(x.new)
  while(abs(test) > tol && iter < max.iter){
    lineSlope <- (fRoot(x.new)-fRoot(x.old))/(x.new-x.old)
    x.old <- x.new
    x.new <- x.new - fRoot(x.new)*(1/lineSlope)
    test <- fRoot(x.new)
    iter <- iter+1
  }
  if(iter == max.iter){
    stop('method did not converge')
  }
  if(print){
    print(x.new)
    print(round(fRoot(x.new), 10))
  }
}
```

Now test the secant method with our test function f(x) = cos(x) - x and calculate the processing time.

```r
secant(fTest, x.old = -10, x.new = 10)
```

```
## [1] 0.7390851
## [1] 0
```

```r
#Single run produces time 0, so run many times
secantTime <- system.time(replicate(1000, secant(fTest, x.old = -10, x.new = 10, print=FALSE)))
secantTime
```

```
##    user  system elapsed
##   0.042   0.004   0.047
```

For comparison, we use the following function for the Newton-Raphson method:

```r
newtonRaphson <- function(fRoot, fDeriv, x.guess, tol = 1e-10, max.iter = 1000, print = TRUE){
  iter <- 1
  test <- fRoot(x.guess)
  while(abs(test) > tol & iter < max.iter){
    x.guess <- x.guess - fRoot(x.guess)/fDeriv(x.guess)
    test <- fRoot(x.guess)
    iter <- iter + 1
  }
  if(iter == max.iter){
    stop('method did not converge')
  }
  if(print){
    print(x.guess)
    print(round(fRoot(x.guess),10))
  }
}
```

Testing and timing the Newton-Raphson method gives:

```
newtonRaphson(fTest, fDerivTest, x.guess = 10)
```

```
## [1] 0.7390851
## [1] 0
```

```
#Single run produces time 0, so run many times
newtonTime <- system.time(replicate(1000, newtonRaphson(fTest, fDerivTest, x.guess = 10, print=FALSE)))
newtonTime
```

```
##    user  system elapsed
##   0.154   0.003   0.159
```

Compare the processing time for the secant and Newton-Raphson methods:

```
secantTime - newtonTime
```

```
##    user  system elapsed
##  -0.112   0.001  -0.112
```

Since the elapsed time is negative, then the secant method took less time to run than the Newton-Raphson. So the secant method is faster by just overs .1 seconds.

Question 2

18 points

The game of craps is played as follows. 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:

```
    x <- sum(ceiling(6*runif(2)))
```

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)

```
craps <- function(numGames = 1, RNGseed = NULL, printRoll = TRUE, printResult = TRUE){
  set.seed(RNGseed)
  #Initialize vector to hold result of games
  winLoss <- rep(NA, numGames)
  #Repeat process 'numGames' times
  for(i in 1:numGames){
    #Compute sum of roll of two dice
    rollSum <- sum(sample(1:6, 2))
    if(printRoll) print(rollSum)
    #Win if sum is 7 or 11
```

```r
    if(rollSum == 7 |rollSum == 11) output <- 'WIN'
    else {
    #If sum is not 7 or 11, store sum value from initial roll and roll again
      rollNext <- sum(sample(1:6, 2))
      while(TRUE){
        if(printRoll) print(rollNext)
        #Lose if 7 or 11 is rolled
        if(rollNext == 7 | rollNext == 11) {
          output <- 'LOSE'
          break
        #Win if same sum is rolled
        } else if(rollNext == rollSum) {
          output <- 'WIN'
          break
        }
        else {
        #If sum is not 7, 11, or original sum, roll again
          rollNext <- sum(sample(1:6, 2))
        }
      }
    }
    if(printResult) print(output)
    winLoss[i] <- output
  }
  #GLobally define final vector of game outcomes
  winLoss <<- winLoss
}
```

Now that we have our craps function, we run it three times with `set.seed(100)`.

```r
#Game 1
craps(RNGseed = 100)
```

```
## [1] 8
## [1] 5
## [1] 9
## [1] 7
## [1] "LOSE"
```

```r
#Game 2
craps(RNGseed = 100)
```

```
## [1] 8
## [1] 5
## [1] 9
## [1] 7
## [1] "LOSE"
```

```r
#Game 3
craps(RNGseed = 100)
```

```
## [1] 8
## [1] 5
## [1] 9
## [1] 7
## [1] "LOSE"
```

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. (5 points)

To find a seed that wins its first 10 games straight, we will run our `craps()` function ten times for a variety of seeds. The `winLoss` vector obtained from these runs will be compared to a vector containing ten instances of "WIN".

```
#Create vector of 10 "WIN"s to compare to winLoss vec for each seed
winTen <- rep("WIN", 10)

#Function to find 10-win seed
findSeed <- function(seedRange){
  for(i in seedRange){
    #Play craps 10 times for each seed, do not print rolls or results
    winLoss <- craps(numGames = 10, RNGseed = i, printRoll = FALSE, printResult = FALSE)
    #Compare with winTen to see if all 10 games were won
    if(identical(winTen, winLoss)) {
      #If 10 wins, print seed and stop
      print(i)
      break
    }
  }
}

findSeed(seedRange = 1:1000)
```

```
## [1] 572
```

So `set.seed(572)` should produce ten winning games of craps. We run the `craps()` function ten times with the seed set to 572.

```
craps(numGames = 10, RNGseed = 572)
```

```
## [1] 7
## [1] "WIN"
## [1] 7
## [1] "WIN"
## [1] 7
## [1] "WIN"
## [1] 5
## [1] 9
## [1] 9
## [1] 5
## [1] "WIN"
## [1] 7
## [1] "WIN"
## [1] 5
## [1] 6
## [1] 6
## [1] 9
## [1] 3
## [1] 4
## [1] 5
## [1] "WIN"
## [1] 3
## [1] 3
## [1] "WIN"
## [1] 7
## [1] "WIN"
## [1] 10
## [1] 4
## [1] 6
## [1] 10
## [1] "WIN"
## [1] 7
## [1] "WIN"
```

Question 3

12 points

Obtain a copy of the football-values lecture. Save the five 2015 CSV files in your working directory.

Modify the code to create a function. This function will create dollar values given information (as arguments) about a league setup. It will return a data.frame and write this data.frame to a CSV file. The final data.frame should contain the columns 'PlayerName', 'pos', 'points', 'value' and be orderd by value descendingly. Do not round dollar values.

Note that the returned data.frame should have `sum(posReq)*nTeams` rows.

Define the function as such (6 points):

```r
# path: directory path to input files
# file: name of the output file; it should be written to path
# nTeams: number of teams in league
# cap: money available to each team
# posReq: number of starters for each position
# points: point allocation for each category
ffvalues <- function(path, file='outfile.csv', nTeams=12, cap=200, posReq=c(qb=1, rb=2, wr=3, te=1, k=1
                     points=c(fg=4, xpt=1, pass_yds=1/25, pass_tds=4, pass_ints=-2,
                              rush_yds=1/10, rush_tds=6, fumbles=-2, rec_yds=1/20, rec_tds=6)) {
  ## read in CSV files
  ## calculate dollar values
  ## save dollar values as CSV file
  ## return data.frame with dollar values
}
```

Define the function as required:

```r
ffvalues <- function(path = "~/Documents/BIOS 6301/Homework/", file = 'outfile.csv', nTeams=12, cap=200
  #Read in files
  k <- read.csv('proj_k15.csv', header=TRUE, stringsAsFactors=FALSE)
```

```r
qb <- read.csv('proj_qb15.csv', header=TRUE, stringsAsFactors=FALSE)
rb <- read.csv('proj_rb15.csv', header=TRUE, stringsAsFactors=FALSE)
te <- read.csv('proj_te15.csv', header=TRUE, stringsAsFactors=FALSE)
wr <- read.csv('proj_wr15.csv', header=TRUE, stringsAsFactors=FALSE)

# generate unique list of column names
cols <- unique(c(names(k), names(qb), names(rb), names(te), names(wr)))

#Add column for position
k[,'pos'] <- 'k'
qb[,'pos'] <- 'qb'
rb[,'pos'] <- 'rb'
te[,'pos'] <- 'te'
wr[,'pos'] <- 'wr'

# append 'pos' to unique column list
cols <- c(cols, 'pos')

# create common columns in each data.frame
# initialize values to zero
k[,setdiff(cols, names(k))] <- 0
qb[,setdiff(cols, names(qb))] <- 0
rb[,setdiff(cols, names(rb))] <- 0
te[,setdiff(cols, names(te))] <- 0
wr[,setdiff(cols, names(wr))] <- 0

# combine data.frames by row, using consistent column order
x <- rbind(k[,cols], qb[,cols], rb[,cols], te[,cols], wr[,cols])

# calculate new columns
# convert NFL stat to fantasy points
x[,'p_fg'] <- x[,'fg']*points["fg"]
x[,'p_xpt'] <- x[,'xpt']*points["xpt"]
x[,'p_pass_yds'] <- x[,'pass_yds']*points["pass_yds"]
x[,'p_pass_tds'] <- x[,'pass_tds']*points["pass_tds"]
x[,'p_pass_ints'] <- x[,'pass_ints']*points["pass_ints"]
x[,'p_rush_yds'] <- x[,'rush_yds']*points["rush_yds"]
x[,'p_rush_tds'] <- x[,'rush_tds']*points["rush_tds"]
x[,'p_fumbles'] <- x[,'fumbles']*points["fumbles"]
x[,'p_rec_yds'] <- x[,'rec_yds']*points["rec_yds"]
x[,'p_rec_tds'] <- x[,'rec_tds']*points["rec_tds"]

# sum selected column values for every row
# this is total fantasy points for each player
x[,'points'] <- rowSums(x[,grep("^p_", names(x))])

# create new data.frame ordered by points descendingly
x2 <- x[order(x[,'points'], decreasing=TRUE),]

# determine the row indeces for each position
k.ix <- which(x2[,'pos']=='k')
qb.ix <- which(x2[,'pos']=='qb')
rb.ix <- which(x2[,'pos']=='rb')
```

```r
te.ix <- which(x2[,'pos']=='te')
wr.ix <- which(x2[,'pos']=='wr')

# calculate marginal points by subtracting "baseline" player's points

if(posReq["k"] == 0){
  x2[k.ix, 'marg'] <- 0
} else{
  x2[k.ix, 'marg'] <- x2[k.ix,'points'] - x2[k.ix[nTeams*posReq["k"]],'points']
}
if(posReq["qb"] == 0){
  x2[qb.ix, 'marg'] <- 0
} else{
  x2[qb.ix, 'marg'] <- x2[qb.ix,'points'] - x2[qb.ix[nTeams*posReq["qb"]],'points']
}
if(posReq["rb"] == 0){
  x2[rb.ix, 'marg'] <- 0
} else{
  x2[rb.ix, 'marg'] <- x2[rb.ix,'points'] - x2[rb.ix[nTeams*posReq["rb"]],'points']
}
if(posReq["te"] == 0){
  x2[te.ix, 'marg'] <- 0
} else{
  x2[te.ix, 'marg'] <- x2[te.ix,'points'] - x2[te.ix[nTeams*posReq["te"]],'points']
}
if(posReq["wr"] == 0){
  x2[wr.ix, 'marg'] <- 0
} else{
  x2[wr.ix, 'marg'] <- x2[wr.ix,'points'] - x2[wr.ix[nTeams*posReq["wr"]],'points']
}


# create a new data.frame subset by non-negative marginal points
x3 <- x2[x2[,'marg'] >= 0,]

# re-order by marginal points
x3 <- x3[order(x3[,'marg'], decreasing=TRUE),]

# reset the row names
rownames(x3) <- NULL

# calculation for player value
x3[,'value'] <- x3[,'marg']*(nTeams*cap-nrow(x3))/sum(x3[,'marg']) + 1

for(i in 1:length(posReq)){
  if(posReq[i] == 0){
    x3 <- x3[!x3[,'pos'] == names(posReq)[i],]
  }
}

dataFrame <- x3[,c('PlayerName','pos','points','value')]

file <- write.csv(dataFrame, file=file, row.names=FALSE)
```

```r
  return(dataFrame)
}
```

1. Call `x1 <- ffvalues('.')`

```r
x.1 <- ffvalues('.')
```

```
1.  How many players are worth more than $20? (1 point)
```

```r
higher20 <- x.1[which(x.1[,"value"] > 20),]
nrow(higher20)
```

```
## [1] 40
```

```
1.  Who is 15th most valuable running back (rb)? (1 point)
```

```r
rbs <- x.1[which(x.1[,"pos"] == "rb"),]
rbs[15,]
```

```
##       PlayerName pos points    value
## 34 Melvin Gordon  rb 152.57 27.59549
```

1. Call `x2 <- ffvalues(getwd(), '16team.csv', nTeams=16, cap=150)`

```r
x.2 <- ffvalues(getwd(), '16team.csv', nTeams=16, cap=150)
```

```
1.  How many players are worth more than $20? (1 point)
```

```r
higher20 <- x.2[which(x.2[,"value"] > 20),]
nrow(higher20)
```

```
## [1] 41
```

```
1.  How many wide receivers (wr) are in the top 40? (1 point)
```

```r
top40 <- x.2[1:40,]
wrTop40 <- top40[which(top40[,'pos'] == 'wr'),]
nrow(wrTop40)
```

```
## [1] 13
```

1. Call: `x3 <- ffvalues('.', 'qbheavy.csv', posReq=c(qb=2, rb=2, wr=3, te=1, k=0), points=c(fg=0, xpt=0, pass_yds=1/25, pass_tds=6, pass_ints=-2, rush_yds=1/10, rush_tds=6, fumbles=-2, rec_yds=1/20, rec_tds=6))`

```
x.3 <- ffvalues('.', 'qbheavy.csv', posReq=c(qb=2, rb=2, wr=3, te=1, k=0),
             points=c(fg=0, xpt=0, pass_yds=1/25, pass_tds=6, pass_ints= -2, rush_yds=1/10, rush_tds=6,
```

```
1.  How many players are worth more than $20? (1 point)
```

```
higher20 <- x.3[which(x.3[,"value"] > 20),]
nrow(higher20)
```

```
## [1] 44
```

```
1.  How many quarterbacks (qb) are in the top 30? (1 point)
```

```
top30 <- x.3[1:30,]
qbTop30 <- top30[which(top30[,'pos'] == 'qb'),]
nrow(qbTop30)
```

```
## [1] 13
```

Question 4

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)

```
numArgs <- rep(NA, length(funs))
for(i in 1:length(funs)){
  numArgs[i] <- length(formals(fun = names(funs)[i]))
}
index <- which.max(numArgs)
funs[index]
```

```
## $scan
## function (file = "", what = double(), nmax = -1L, n = -1L, sep = "",
##     quote = if (identical(sep, "\n")) "" else "'\"", dec = ".",
##     skip = 0L, nlines = 0L, na.strings = "NA", flush = FALSE,
##     fill = FALSE, strip.white = FALSE, quiet = FALSE, blank.lines.skip = TRUE,
##     multi.line = TRUE, comment.char = "", allowEscapes = FALSE,
##     fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
## {
##     na.strings <- as.character(na.strings)
##     if (!missing(n)) {
##         if (missing(nmax))
##             nmax <- n/pmax(length(what), 1L)
##         else stop("either specify 'nmax' or 'n', but not both.")
##     }
##     if (missing(file) && !missing(text)) {
##         file <- textConnection(text, encoding = "UTF-8")
##         encoding <- "UTF-8"
##         on.exit(close(file))
##     }
##     if (is.character(file))
##         if (file == "")
##             file <- stdin()
##         else {
##             file <- if (nzchar(fileEncoding))
##                 file(file, "r", encoding = fileEncoding)
##             else file(file, "r")
##             on.exit(close(file))
##         }
##     if (!inherits(file, "connection"))
##         stop("'file' must be a character string or connection")
##     .Internal(scan(file, what, nmax, sep, dec, quote, skip, nlines,
##         na.strings, flush, fill, strip.white, quiet, blank.lines.skip,
##         multi.line, comment.char, allowEscapes, encoding, skipNul))
## }
## <bytecode: 0x7fdec3125158>
## <environment: namespace:base>
```

```
numArgs[index]
```

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

So the `scan()` function has the most, with 22 arguments.

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

```
noArgs <- rep(0, length(funs))
for(i in 1:length(funs)){
  if(length(formals(fun = names(funs)[i])) == 0) noArgs[i] <- 1
}
sum(noArgs)
```

```
## [1] 225
```

There are 225 functions with no arguments.

Source: `formals()` function obtained from http://adv-r.had.co.nz/Functions.html

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