Bios 6301: Assignment 4

Andrea Perreault

Grade 46/50

Due Tuesday, 01 November, 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 $f'(x) = -\sin(x) - 1$.

```
# secant method
secant <- function(g1, g2, f, tol=10e-7, maxiter=1000) {
  i <- 0
  while(abs(g1-g2) > tol && i < maxiter) {
    x <- g2 - f(g2)*(g2-g1)/(f(g2)-f(g1))
    g2 <- g1
    x <- g2
    i <- i + 1
  }
```

```
  if(i==maxiter) {
    print('failed to converge')
    return(NULL)
  } else {
    print(sprintf('converges at %s after %s iterations', x, i))
  }
}

f <- function(x) cos(x) - x

# compare to Newton-Raphson method
newton <- function(guess, f, fp, tol=10e-7, maxiter=1000) {
  i <- 1 # counter
  while(abs(f(guess)) > tol && i < maxiter) {
    guess <- guess - f(guess)/fp(guess)
    i <- i + 1
  }
  if(i==maxiter) {
    print('failed to converge')
    return(NULL)
  } else {
    print(sprintf('converges at %s after %s iterations', guess, i))
  }
}

f <- function(x) cos(x) - x
fp <- function(x) (-sin(x))-1
```

```
# test 1
(system.time(replicate(1000, secant(0.5, 0.75, f))))
(system.time(replicate(1000, newton(0.5, f, fp))))
# test 2
(system.time(replicate(1000, secant(0, 0.2, f))))
(system.time(replicate(1000, newton(0.25, f, fp))))
```

```
The Newton-Raphson method is faster, taking 0.155 seconds to complete 1000 replicates.
This is compared to 0.189 seconds for the secant method. However, the amount of time it
takes to find the root for these methods varies depending on the guesses input by the
user. For example, it takes 0.175 seconds for the secant method to find the root if the
user gives 0 and 0.2 as initial guesses, compared to 0.210 with the Newton-Raphson method.
```

JC Grading -1

Comment: Please also provide output showing that the secant and newton rhapson methods converge on the same answer.

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)

```
set.seed(100)

craps <- function(x, numGames) {
  results <- matrix(0, nrow = numGames, ncol = 2, dimnames = list(NULL, c("Win", "Lose")))
  for(i in 1:numGames) {
    win <- FALSE
    lose <- FALSE
    if(x==7||x==11){
    win <- TRUE
    print('Yay! You win :)')
  } else {
    x1 <- 0
    while(x1!=x && x1!=7 && x1!=11){
      print('Cross your fingers and roll again.')
      x1 <- sum(ceiling(6*runif(2)))
      print(x1)
    }
    if(x1==7||x1==11){
      lose <- TRUE
      print("Sorry, you lose :(")
    } else {
      win <- TRUE
      print(sprintf('You rolled %s again, you win!', x))
    }
  }
    if(win) {
    results[i, 1] <- 1
  } else {
    results[i, 2] <- 1
  }
  }
  return(results)
}

x=sum(ceiling(6*runif(2)))
results <- craps(x, 3)
```

```
## [1] "Cross your fingers and roll again."
## [1] 5
## [1] "Cross your fingers and roll again."
## [1] 6
## [1] "Cross your fingers and roll again."
## [1] 8
## [1] "Cross your fingers and roll again."
## [1] 6
## [1] "Cross your fingers and roll again."
## [1] 10
```

```
## [1] "Cross your fingers and roll again."
## [1] 5
## [1] "Cross your fingers and roll again."
## [1] 10
## [1] "Cross your fingers and roll again."
## [1] 5
## [1] "Cross your fingers and roll again."
## [1] 8
## [1] "Cross your fingers and roll again."
## [1] 9
## [1] "Cross your fingers and roll again."
## [1] 9
## [1] "Cross your fingers and roll again."
## [1] 5
## [1] "Cross your fingers and roll again."
## [1] 11
## [1] "Sorry, you lose :("
## [1] "Cross your fingers and roll again."
## [1] 6
## [1] "Cross your fingers and roll again."
## [1] 9
## [1] "Cross your fingers and roll again."
## [1] 9
## [1] "Cross your fingers and roll again."
## [1] 11
## [1] "Sorry, you lose :("
## [1] "Cross your fingers and roll again."
## [1] 6
## [1] "Cross your fingers and roll again."
## [1] 7
## [1] "Sorry, you lose :("
```

```
results
```

```
##      Win Lose
## [1,]   0    1
## [2,]   0    1
## [3,]   0    1
```

2. 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)

```
set.seed(500)

craps <- function(x, numGames) {
  results <- matrix(0, nrow = numGames, ncol = 2, dimnames = list(NULL, c("Win", "Lose")))
  for(i in 1:numGames) {
    win <- FALSE
    lose <- FALSE
    if(x==7||x==11){
    win <- TRUE
  } else {
    x1 <- 0
    while(x1!=x && x1!=7 && x1!=11){
      x1 <- sum(ceiling(6*runif(2)))
    }
```

```
    if(x1==7||x1==11){
      lose <- TRUE
    } else {
      win <- TRUE
    }
  }
    if(win) {
    results[i, 1] <- 1
  } else {
    results[i, 2] <- 1
  }
  }
  return(results)
}

x=sum(ceiling(6*runif(2)))
results <- craps(x, 10)
results
```

```
##       Win Lose
##  [1,]   1    0
##  [2,]   1    0
##  [3,]   1    0
##  [4,]   1    0
##  [5,]   1    0
##  [6,]   1    0
##  [7,]   1    0
##  [8,]   1    0
##  [9,]   1    0
## [10,]   1    0
```

Question 3

12 points

Obtain a copy of the football-values lecture. Save the five 2016 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):

```
# 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
  # read in files
  k <- read.csv('proj_k16.csv', header=TRUE, stringsAsFactors=FALSE)
```

```
qb <- read.csv('proj_qb16.csv', header=TRUE, stringsAsFactors=FALSE)
rb <- read.csv('proj_rb16.csv', header=TRUE, stringsAsFactors=FALSE)
te <- read.csv('proj_te16.csv', header=TRUE, stringsAsFactors=FALSE)
wr <- read.csv('proj_wr16.csv', header=TRUE, stringsAsFactors=FALSE)

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

# convert NFL stat to fantasy points
x[,'p_fg'] <- x[,'fg']*points[[1]]
x[,'p_xpt'] <- x[,'xpt']*points[[2]]
x[,'p_pass_yds'] <- x[,'pass_yds']*points[[3]]
x[,'p_pass_tds'] <- x[,'pass_tds']*points[[4]]
x[,'p_pass_ints'] <- x[,'pass_ints']*points[[5]]
x[,'p_rush_yds'] <- x[,'rush_yds']*points[[6]]
x[,'p_rush_tds'] <- x[,'rush_tds']*points[[7]]
x[,'p_fumbles'] <- x[,'fumbles']*points[[8]]
x[,'p_rec_yds'] <- x[,'rec_yds']*points[[9]]
x[,'p_rec_tds'] <- x[,'rec_tds']*points[[10]]

# sum selected column values for every row
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.idx <- which(x2[,'pos']=='k')
qb.idx <- which(x2[,'pos']=='qb')
rb.idx <- which(x2[,'pos']=='rb')
te.idx <- which(x2[,'pos']=='te')
wr.idx <- which(x2[,'pos']=='wr')

# calculate marginal points by subtracting "baseline" player's points
x2[k.idx, 'marg'] <- x2[k.idx,'points'] - x2[k.idx[12],'points']
```

```
  x2[qb.idx, 'marg'] <- x2[qb.idx,'points'] - x2[qb.idx[12],'points']
  x2[rb.idx, 'marg'] <- x2[rb.idx,'points'] - x2[rb.idx[24],'points']
  x2[te.idx, 'marg'] <- x2[te.idx,'points'] - x2[te.idx[12],'points']
  x2[wr.idx, 'marg'] <- x2[wr.idx,'points'] - x2[wr.idx[36],'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

  # save dollar values to a csv file
  x4 <- x3[,c('PlayerName','pos','points','value')]
  write.csv(x4, file)
  # return data.frame with dollar values
  as.data.frame(x4, row.names=NULL)
}
```

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

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

```
x1 <- ffvalues('.')
which(x1[,'value']>20)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
```

```
There are 46 players that are worth more than $20.
```

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

```
which(x1[,'pos']=='rb')[15]
```

```
## [1] 47
```

```
The 15th most valuable running back is at position 47, which is Carlos Hyde.
```

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

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

```
x2 <- ffvalues(getwd(), '16team.csv', nTeams=16, cap=150)
which(x2[,'value']>20)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
```

```
There are 46 players that are worth more than $20.
```

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

```
which(x2[,'pos']=='wr')[1:40]
```

```
##  [1]  2  6  9 13 15 18 19 24 25 29 30 31 32 33 34 37 38 40 42 43 44 45 50
```

```
## [24] 51 53 55 59 62 67 71 75 77 84 86 87 95 NA NA NA NA

There are 36 wide receivers in the top 40.
```

JC Grading -1 Comment: which tells you the position of the wide receivers. So, count how many have the position <= 40. There are 18.

3. 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))
```

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

```
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_t
which(x3[,'value']>20)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46

There are 46 players that are worth more than $20.
```

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

```
which(x3[,'pos']=='qb')[1:30]
```

```
##  [1]  1  3  6 11 17 31 39 48 53 57 80 81 NA NA NA NA NA NA NA NA NA NA NA
## [24] NA NA NA NA NA NA NA

There are 12 quarterbacks in the top 30.
```

JC Grading -2

Comment: The answers to the above to questions are 50 and 10, respectively. Look to the commands under comment: # calculate marginal points by subtracting "baseline" player's points. You've hard coded the number of players in the position rather than using the function's input posReq.

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 <- matrix(0, nrow = length(funs), ncol = 2, dimnames = list(NULL, c("Function", "Arguments")))
numArgs <- as.data.frame(numArgs)
x1 <- lapply(funs, function(x) length(formals(x)))
numArgs[,1] <- names(x1)
numArgs[,2] <- as.integer(x1)
numArgs <- numArgs[order(numArgs[,'Arguments'], decreasing=TRUE),]
head(numArgs)
```

```
##             Function Arguments
## 938             scan        22
```

```
## 470   format.default        16
## 483          formatC        14
## 705 merge.data.frame        12
## 793        prettyNum        12
## 987           source        12
```

```
The scan function has 22 arguments.
```

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

```
which(numArgs[,2]==0)
```

```
##   [1]  979  980  981  982  983  984  985  986  987  988  989  990  991  992
##  [15]  993  994  995  996  997  998  999 1000 1001 1002 1003 1004 1005 1006
##  [29] 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020
##  [43] 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034
##  [57] 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048
##  [71] 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062
##  [85] 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076
##  [99] 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090
## [113] 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104
## [127] 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118
## [141] 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132
## [155] 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146
## [169] 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160
## [183] 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174
## [197] 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188
## [211] 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202
## [225] 1203
```

```
There are 225 functions with no arguments.
```