Bios 6301: Assignment 4

Sarah Lotspeich

10/17/2016

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:

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

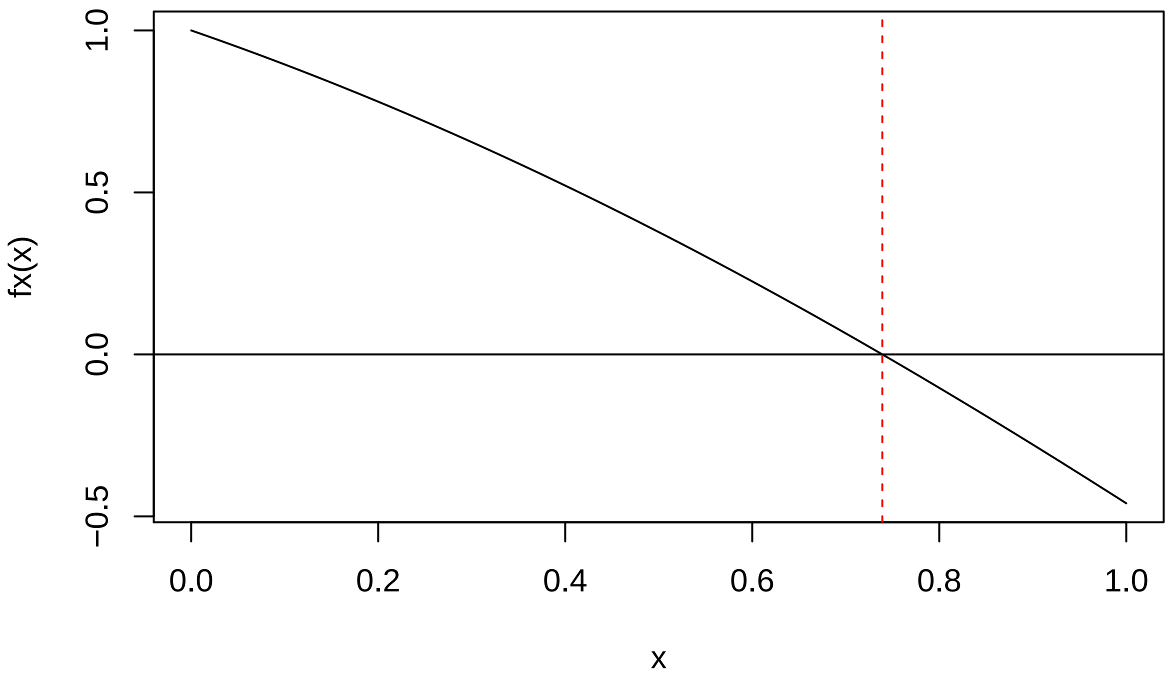
Notice that we no longer need to know f' but in return we have to provide two initial points, x_0 and x_1.

Write a function that implements the secant algorithm. Validate your program by finding the root of the function $f(x) = \cos(x) - x$.

```r
fx <- function(x)
{
  return(cos(x) - x)
}

secantAlg <- function(x0, tol=1e-8, max_iter=100) {
  stopifnot(tol > 0, max_iter > 0)
  x1 <- x0 + 1e6
  iter <- 0
  while(abs(x1-x0) > tol && iter < max_iter) {
    fx0 <- fx(x0)
    fx1 <- fx(x1)
    x2 <- x1 - (fx1*((x1-x0)/(fx1-fx0)))
    x0 <- x1
    x1 <- x2
    iter <- iter + 1
  }
  if(abs(x0-x1) > tol)
  {
    x1 <- NULL
    print("Algorithm failed to converge!")
  }
  return(x1)
}

x2 <- secantAlg(10) #find the root of f(x) = cos(x) + x
curve(fx)
abline(h=0) #plot x axis
abline(v=x2, col="red", lty=2) #plot root
```

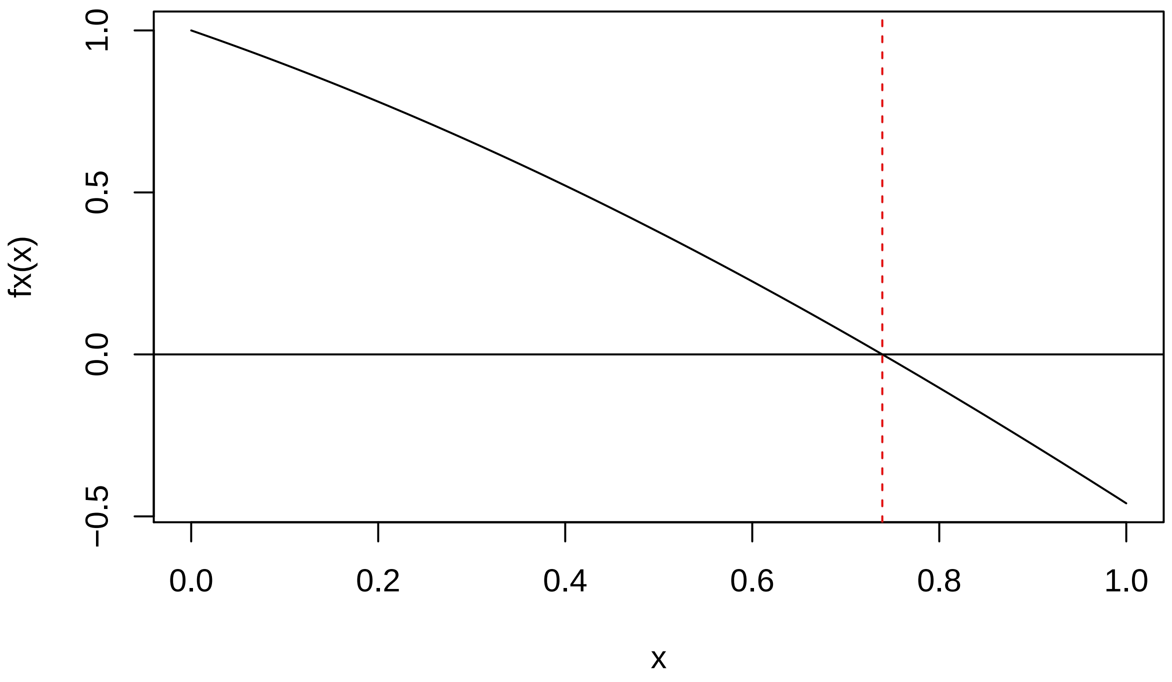


Compare its performance with the Newton-Raphson method – which is faster, and by how much? For this example $f'(x) = -\sin(x) - 1$.

```r
#Newton-Raphson Method
fxdx <- function(x)
{
  return(-sin(x) - 1)
}

newtonRaphsonAlg <- function(x0, max_iter=100) {
  stopifnot(max_iter > 0)
  iter <- 0
  while(iter < max_iter) {
    fx0 <- fx(x0)
    fxdx0 <- fxdx(x0)
    x1 <- x0 - (fx0/fxdx0)
    x0 <- x1
    iter <- iter + 1
  }
  #if (iter == max_iter)
  #{
    #print("Algorithm failed to converge!")
    #return(NULL)
  #}
  #else
  return(x0)
}

x1 <- newtonRaphsonAlg(x0=10)
curve(fx)
abline(h=0) #plot x axis
abline(v=x1, col="red", lty=2) #plot root
```



```r
system.time(replicate(10000,secantAlg(10)))
```

```
##    user  system elapsed
##   0.490   0.012   0.514
```

```r
system.time(replicate(10000,newtonRaphsonAlg(10)))
```

```
##    user  system elapsed
##   3.247   0.025   3.295
```

With fewer inputs, the Newton-Raphson Method returned the exact same root for the equation $f(x) = cos(x) + x$. However, this method is limited by our ability to differentiate the function of interest $f(x)$. Overall, the function to carry out the Newton-Raphson algorithm is much simpler than that of the Secant algorithm, because the latter requires two points and a given amount of tolerance that will continue to move toward each other until capturing the root. When repeating each algorithm 10,000 times and clocking them with system.time(), the Secant Algorithm seemed to blow the Newton-Raphson Algorithm out of the water! The Newton-Raphson Algorithm took more than 3 seconds to execute 10,000 times, whereas the Secant Algorithm accomplished the same task in less than half a second.

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)))
```

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)

```r
set.seed(100) #set seed as specified

crapsGame <- function(output)
{
  keepRolling <- TRUE
  firstRoll <- TRUE
  point <- 0
  while(keepRolling)
  {
  x <- sum(ceiling(6*runif(2))) #simulate dice roll
    if (firstRoll == TRUE)
    {
      if (output == TRUE)
      {
        print("New game.")
        print(paste("Sum of the first roll was",x)) #print first x value
        if (x == 7 || x == 11)
        {
            print("Win!")
            return(print("End of game."))
        }
        else
        {
            print(paste("Roll again with ", point <- x, " as point."))
            firstRoll <- FALSE
        }
      }
      else
      {
        if (x == 7 || x == 11)
        {
          return(1)
        }
        else
        {
          point <- x
          firstRoll <- FALSE
        }
      }
    }
    else
    {
      if (output == TRUE)
      {
        print(paste("Next sum of roll was ",x))
        if (x == point)
        {
          print("Win!")
          return(print("End of game."))
        }
        if (x == 7 || x == 11)
        {
          print("Lose!")
          return(print("End of game."))
        }
      }
      else
      {
        if (x == point)
        {
          return(1)
        }
        if (x == 7 || x == 11)
        {
          return(0)
        }
      }
    }
  }
}

for (i in 1:3)
{
  crapsGame(TRUE)
}
```

```
## [1] "New game."
## [1] "Sum of the first roll was 4"
## [1] "Roll again with  4  as point."
## [1] "Next sum of roll was  5"
## [1] "Next sum of roll was  6"
## [1] "Next sum of roll was  8"
## [1] "Next sum of roll was  6"
## [1] "Next sum of roll was  10"
## [1] "Next sum of roll was  5"
## [1] "Next sum of roll was  10"
## [1] "Next sum of roll was  5"
## [1] "Next sum of roll was  8"
## [1] "Next sum of roll was  9"
## [1] "Next sum of roll was  9"
## [1] "Next sum of roll was  5"
## [1] "Next sum of roll was  11"
## [1] "Lose!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 6"
## [1] "Roll again with  6  as point."
## [1] "Next sum of roll was  9"
## [1] "Next sum of roll was  9"
## [1] "Next sum of roll was  11"
## [1] "Lose!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 6"
## [1] "Roll again with  6  as point."
## [1] "Next sum of roll was  7"
## [1] "Lose!"
## [1] "End of game."
```

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)

```r
seeds <- seq(1,10000)

trySeed <- function(seed)
{
  set.seed(seed)
  wins <- NULL
  for (i in 1:10)
  {
    wins[i] <- crapsGame(FALSE)
  }
  return(sum(wins)) #return wins for that seed
}

winsVector <- sapply(seeds,trySeed)
compSeeds <- as.data.frame(cbind(seeds,winsVector))
which(compSeeds$winsVector == 10) #find seeds between 1 and 10,000 that will yield 10 wins in a row
```

```
## [1]  880 1639 4352 4411 8839 9085
```

```r
set.seed(880) #display output for first seed with 10 straight wins
for (i in 1:10)
{
  crapsGame(TRUE)
}
```

```
## [1] "New game."
## [1] "Sum of the first roll was 7"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 8"
## [1] "Roll again with  8  as point."
## [1] "Next sum of roll was  9"
## [1] "Next sum of roll was  3"
## [1] "Next sum of roll was  10"
## [1] "Next sum of roll was  6"
## [1] "Next sum of roll was  8"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 10"
## [1] "Roll again with  10  as point."
## [1] "Next sum of roll was  10"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 9"
## [1] "Roll again with  9  as point."
## [1] "Next sum of roll was  9"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 11"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 8"
## [1] "Roll again with  8  as point."
## [1] "Next sum of roll was  8"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 5"
## [1] "Roll again with  5  as point."
## [1] "Next sum of roll was  5"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 7"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 9"
## [1] "Roll again with  9  as point."
## [1] "Next sum of roll was  9"
## [1] "Win!"
## [1] "End of game."
## [1] "New game."
## [1] "Sum of the first roll was 7"
## [1] "Win!"
## [1] "End of game."
```

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):

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 } Call x1 <- ffvalues('.')

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

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

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

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

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

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

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

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.

Which function has the most arguments? (3 points)

How many functions have no arguments? (2 points)

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