Bios 6301: Assignment 5

Max Rohde

```r
library(tidyverse)
```

```
## -- Attaching packages -------------------------------------------------------------------------------
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
## -- Conflicts ----------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
library(microbenchmark)
```

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 $f'(x) = -\sin(x) - 1$.

```r
### Newton-Raphson Method
# Arguments
# - Initial guess
# - The function of which to find the root
# - The derivative of the function
```

```r
newton_raphson <- function(x, f, f_prime) {
  while (abs(f(x)) > 1e-8)
  {
    # Update x by linear approximation
    x <- (x - (f(x) / f_prime(x)))
  }
  return(x)
}

### Secant Method
# Arguments
# - First guess
# - Second guess
# - The function of which to find the root
secant_method <- function(x0, x1, f) {
  while (abs(f(x1)) > 1e-8)
  {
    # Calculate slope between the points
    m <- (f(x0) - f(x1)) / (x0-x1)

    # Set x1 to x0
    x0 <- x1

    # Solve for new x1 with linear approximation
    x1 <- (x1 - (f(x1) / m))
  }
  return(x1)
}

# Compare the Newton-Raphson method to the Secant method.
# Both give the same solution and agree with the built-in
# uniroot() function
newton_raphson(1, function(x) cos(x) - x, function(x) -sin(x) - 1)
```

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

```r
secant_method(1, 2, function(x) cos(x) - x)
```

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

```r
uniroot(function(x) cos(x) - x, interval=c(-50,50), tol = 1e-8)$root
```

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

```r
# Time the two functions for 1e5 iterations each
timing <- microbenchmark(
  newton_raphson(1, function(x) cos(x) - x, function(x) -sin(x) - 1),
  secant_method(1, 2, function(x) cos(x) - x),
  times=1e5
)

timing
```

```
## Unit: microseconds
##                                                                   expr   min
##  newton_raphson(1, function(x) cos(x) - x, function(x) -sin(x) -      1) 4.186
```

```
##                              secant_method(1, 2, function(x) cos(x) - x) 7.266
##     lq     mean median    uq       max neval
##  5.250  5.91221  5.465 5.853  2884.685 1e+05
##  9.076 10.11932  9.395 9.858 12344.298 1e+05
```

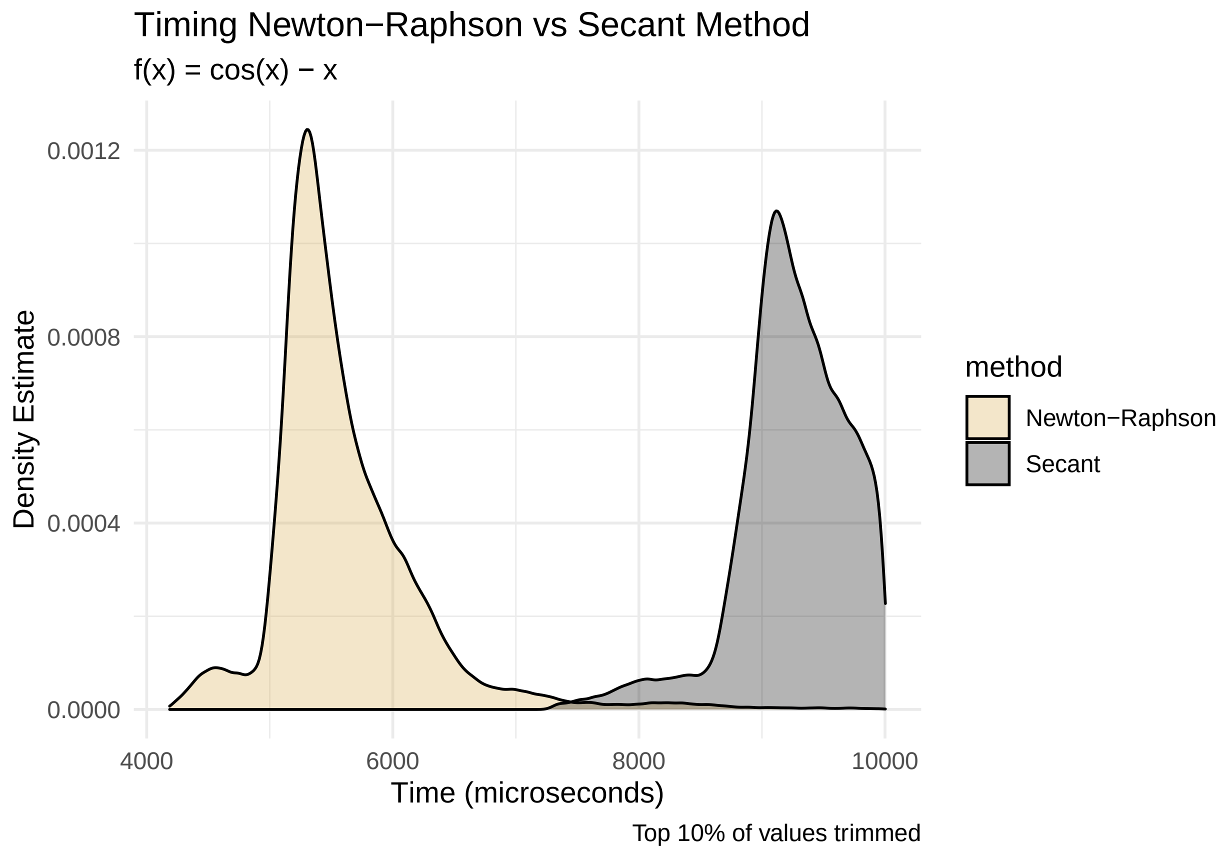
We see that the Newton-Raphson method is faster on average, with a median time of 5.5 microseconds compared to 9.3 microseconds for the Secant method.

The plots below show the distribution of times.

```r
# Put the timing data into a data frame
df <- tibble(time = timing$time, method = timing$expr)

# Rename to nicer names for plotting
df$method <- recode(
        df$method,
        `newton_raphson(1, function(x) cos(x) - x, function(x) -sin(x) -      1)` = "Newton-Raphson",
       `secant_method(1, 2, function(x) cos(x) - x)` = "Secant"
       )

# Plot the data using a KDE plot
df %>%
  filter(time < quantile(df$time, 0.90)) %>%
  ggplot() +
  geom_density(aes(x = time, fill = method), alpha=0.3) +
  scale_fill_manual(values= c("#D8AB4C", "#000000"))+
  labs(title="Timing Newton-Raphson vs Secant Method",
       subtitle="f(x) = cos(x) - x",
       caption="Top 10% of values trimmed",
       x="Time (microseconds)", y="Density Estimate",
       color="")+
  theme_minimal()
```



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:

```
# Returns the roll of two dice rolls
# and prints the value of the roll
roll <- function(){
  roll <- sum(ceiling(6*runif(2)))
  print(paste("You rolled ", roll))
  return(roll)
  }
```

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)

```
# Simulate a simplified game of craps
# Return 1 if Win, 0 if Lose
craps <- function(no_output=FALSE){
  first_roll <- roll()
  if(first_roll %in% c(7,11)){
    print("YOU WIN!")
```

```r
    return(1)
  }
  else{
    while(TRUE){
      x <- roll()
      if(x==first_roll){
        print("YOU WIN!")
        return(1)
      }
      else if(x %in% c(7,11)){
        print("YOU LOSE!")
        return(0)
      }
    }
  }
}

# Creates a silent version of craps() using purrr:quietly()
# The output is stored in $result
quiet_craps <- quietly(craps)

# Test on three games
set.seed(100)
for (i in 1:3){
  print(paste("Game ", i))
  craps()
  }
```

```
## [1] "Game  1"
## [1] "You rolled  4"
## [1] "You rolled  5"
## [1] "You rolled  6"
## [1] "You rolled  8"
## [1] "You rolled  6"
## [1] "You rolled  10"
## [1] "You rolled  5"
## [1] "You rolled  10"
## [1] "You rolled  5"
## [1] "You rolled  8"
## [1] "You rolled  9"
## [1] "You rolled  9"
## [1] "You rolled  5"
## [1] "You rolled  11"
## [1] "YOU LOSE!"
## [1] "Game  2"
## [1] "You rolled  6"
## [1] "You rolled  9"
## [1] "You rolled  9"
## [1] "You rolled  11"
## [1] "YOU LOSE!"
## [1] "Game  3"
## [1] "You rolled  6"
## [1] "You rolled  7"
## [1] "YOU 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. (7 points)

```
seed <- 0
while(TRUE){
  set.seed(seed)
  results <- map_dbl(1:10, ~quiet_craps()$result) # Simulate 10 games
  if (sum(results) == 10) {
    print(seed)
    break
  }
  else{
    seed <- seed + 1
  }
}
```

```
## [1] 880
```

We see that the seed 880 will produce 10 wins in a row.

```
# Verify that the seed produces 10 wins in a row
set.seed(880)
map_dbl(1:10, ~craps())
```

```
## [1] "You rolled  7"
## [1] "YOU WIN!"
## [1] "You rolled  8"
## [1] "You rolled  9"
## [1] "You rolled  3"
## [1] "You rolled  10"
## [1] "You rolled  6"
## [1] "You rolled  8"
## [1] "YOU WIN!"
## [1] "You rolled  10"
## [1] "You rolled  10"
## [1] "YOU WIN!"
## [1] "You rolled  9"
## [1] "You rolled  9"
## [1] "YOU WIN!"
## [1] "You rolled  11"
## [1] "YOU WIN!"
## [1] "You rolled  8"
## [1] "You rolled  8"
## [1] "YOU WIN!"
## [1] "You rolled  5"
## [1] "You rolled  5"
## [1] "YOU WIN!"
## [1] "You rolled  7"
## [1] "YOU WIN!"
## [1] "You rolled  9"
## [1] "You rolled  9"
## [1] "YOU WIN!"
## [1] "You rolled  7"
## [1] "YOU WIN!"
```

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

Question 3

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

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

```
# A function that takes a function and returns the number of arguments
num_args <- function(f){
  arg_list <- as.list(args(f))
  return(length(arg_list))
}

# Create a named vector with the number of arguments in every function
arguments <- map_dbl(funs, ~num_args(.x))

# Find function with the most arguments
arguments %>%
  sort(decreasing = TRUE) %>%
  head()
```

```
##            scan   format.default           source          formatC
##              23               17               17               16
##         library merge.data.frame
##              14               14
```

```
# Get number of functions with no arguments
(arguments == 0) %>% sum()
```

```
## [1] 25
```

The `scan` function has the most arguments at 23 arguments.

There are 25 functions in `base` with no arguments.