Homework 4 – Due March 24

Name: Navya Mittal netID: navya_0215 Collaborated with: Navya Mittal

Your homework **must be submitted in Word or PDF format, created by calling "Knit Word" or "Knit PDF" from RStudio on your R Markdown document. Submission in other formats may receive a grade of 0**. Your responses must be supported by both textual explanations and the code you generate to produce your result. Note that all R code used to produce your results must be shown in your knitted file.

Simulation

1. In this question, you will use simulation to estimate the value of π based on the fact that the area of circle with radius 1 is π.

a. Set the seed of the random number generator to `1234`

```
set.seed(1234)
```

b. Generate $X_1, \ldots, X_n, Y_1, \ldots, Y_n$ as i.i.d. random variables from $Uniform(-1, 1)$, where $n = 100$. **Don't display these 200 random variables in your solution.**

```
n <- 100
X <- runif(n, -1, 1)
Y <- runif(n, -1, 1)
```

c. Regard (X_i, Y_i) as an coordinate in the xy plane. Find out the number of pairs (X_i, Y_i)'s that fall within a circle centered at $(0, 0)$ with radius 1.

```
num_pairs_in_circle <- 0
for (i in 1:n) {
  distance <- sqrt(X[i]^2 + Y[i]^2)
  if (distance <= 1) {
    num_pairs_in_circle <- num_pairs_in_circle + 1
  }
}

print(num_pairs_in_circle)
```

```
## [1] 76
```

d. Compute the proportion p of pairs that fall within the circle. According to law of large numbers, $4p$ "converges" to π, when as n goes to infinity. Compute the absolute error between $4p$ and π.

```
p <- num_pairs_in_circle / n
pi_estimate <- 4 * p

# print(paste0("Proportion of pairs within circle: ", p))
# print(paste0("Estimate of pi: ", pi_estimate))

error <- abs(pi_estimate - pi)

print(paste0("Absolute error: ", error))
```

```
## [1] "Absolute error: 0.101592653589793"
```

e. Set the seed to `5678`, and repeat part (b)-(d) with $n = 10000$.

```
set.seed(5678)
n=10000
X <- runif(n, -1, 1)
Y <- runif(n, -1, 1)
num_pairs_in_circle <- 0
for (i in 1:n) {
  distance <- sqrt(X[i]^2 + Y[i]^2)
  if (distance <= 1) {
    num_pairs_in_circle <- num_pairs_in_circle + 1
  }
}

print(num_pairs_in_circle)
```

```
## [1] 7826
```

```
p <- num_pairs_in_circle / n
pi_estimate <- 4 * p

# print(paste0("Proportion of pairs within circle: ", p))
# print(paste0("Estimate of pi: ", pi_estimate))

error <- abs(pi_estimate - pi)

print(paste0("Absolute error: ", error))
```

```
## [1] "Absolute error: 0.0111926535897933"
```

Browser

Below is a function `add.up.inv.powers()` that computes $1^1 + 2^{1/2} + \ldots + (n-1)^{1/(n-1)} + n^{1/n}$, via a `for()` loop, for some value of n, specified in the first argument. The second argument is `verbose`; if this is TRUE (the default is FALSE), then the function prints out the current summand to the console, as a roman numeral. A short demo is given below. You'll use `add.up.inv.powers()` and `roman.cat()` to do a bit of exploration with `browser()` in the next several questions. Remember to use `eval = FALSE` for chunks that call browser.

```
add.up.inv.powers = function(n, verbose=FALSE) {
  x = 0
  for (i in 1:n) {
    x = x + i^(1/i)
    if (verbose) roman.cat(i)
  }
  if (verbose) cat("\n")
  return(x)
}

roman.cat = function(num) {
  roman.num = as.roman(num)
  roman.str = as.character(roman.num)
  cat(roman.str, "... ")
}
add.up.inv.powers(n=3, verbose=FALSE)
```

```
## [1] 3.856463
```

```
add.up.inv.powers(n=5, verbose=FALSE)
```

```
## [1] 6.650406
```

```
add.up.inv.powers(n=10, verbose=FALSE)
```

```
## [1] 13.15116
```

2. Copy and paste the definition of `add.up.inv.powers()` below, into an R code chunk that will *not* be evaluated when you knit (hence the `eval=FALSE`). You'll use this as a working ground for the code that you'll run in your console. Place a call to `browser()` inside `add.up.inv.powers()`, in between the line `x = 0` and the `for()` loop. Then update this function definition in your console (i.e., just run the code block that defines `add.up.inv.powers()`), and call the function in the console with `n=5` and the default value of `verbose`.

```
add.up.inv.powers = function(n, verbose=FALSE) {
  x = 0
  browser()
  for (i in 1:n) {
    x = x + i^(1/i)
    if (verbose) roman.cat(i)
  }

  if (verbose) cat("\n")
  return(x)
}

#add.up.inv.powers(5) ran this in the console
```

Answer the following questions, exploring what you can do in browser mode.

a. How do you display the value of the variable `n` defined in the `add.up.inv.powers()` function? (Recall that typing "n" just gives you the next line.)

We can type in print(n) in the console and it gives the value of n, that is, 5 in this case.

b. How do you exit the browser mode prematurely, before the last line is reached?

We can type in "Q" to stop debugging and return to the console or use the red squre stop button in the taskbar of the console.

c. Suppose you were to run in the browser mode a call like `cool.new.num = add.up.inv.powers(n=5)` in the console; if you ran the browser to completion (i.e., stepped through all the lines of the function until a return), would the variable `cool.new.num` be defined in your console?

Yes, we can see that the variable is defined in the console and upon completion it returns the value of x.

d. What happens if you were to save the output again in a different variable name, but you didn't run the browser to completion, i.e., you exited prematurely?

I entered "out <- add.up.inv.powers(n=5)" in the console and then we enter browser mode. If you don't run it all the way, the variable "out" doesnt get defined or stored with the output of the function. The output of the function is stored in the variable only when its ran all the way.

e. Can you define new variables while in browser mode?

Yes. we can define new variable in browser mode. new_var <- 10

f. Can you redefine existing variables in the browser? What happens, for example, if you were to redefine `x` the moment you entered the browser mode?

It redefines the variable which changes the output.

g. What happens if you change the location of the call to `browser()` within the definition of the function `add.up.inv.powers()`?

It changes the environment of the function. It affects the flow of the variable.

h. Typing the "f" key in browser mode, as soon as you enter a `for()` loop, will skip to the end of the loop. Try this a few times. What happens if you type "f" after say a few iterations of the loop? What happens if you type "f" right before the loop?

It exits the browser and returns the original outcome if we type it soon as it enters the for loop. After a couple of iterations„ if we type f then its gonna finish the loop, go through the entire function and return the outcome. If you type it right before the loop, the browser will skip the loop and executes the lines after the code and returns the original outcome.

i. Lastly, typing the "s" key in browser mode will put you into an even more in-depth mode, call it "follow-the-rabit-hole" mode, where you step into each function being evaluated, and enter browser mode for that function. Try this, and describe what you find. Do you step into `roman.cat()`? Do you step into functions that are built-in? How far down the rabbit hole do you go?

The "s" allows you to run the for loop line by line and debug. Then, it steps into roman.cat() and then it steps into built in function.Then it goes over the lines of the built-in function and it keeps looping going into the source code of each function.

Debugging

In the following questions, you will debug each of the functions included in the corresponding questions so that they produce the correct results. After correcting those problems, you need to show the debugged function and display the result of the testing code based on your debugged function (change the test chunk option to eval = TRUE). Also you have to explain in words what changes you made and why. For convenience, the functions and testing code are also provided in the accompanying code chunks. We are using the `testthat` package for the tests for the tests. Make sure to install the package. Remember to use `eval = FALSE` for chunks that call browser.

3. `my.dgamma.log()` generates the log of the probability density of the Gamma distribution. Use `?lgamma` to understand what it does. Note that this is exactly equal to the function `dgamma(..., log=TRUE)` so use this to verify your answers. (Hint: Tests can pass with 2 changes)
   The function to be debugged:

```
my.dgamma.log <- function(xx, shape, rate) {
  return(shape*log(rate) - lgamma(shape) + (shape-1)*log(xx) - rate*xx)
}
```

I followed the formula of a pdf of a gamma distribution and used log on it to generate the right formula. Then, I corrected the code based on the equation that I attained. I changed shape-1)*xx to (shape-1)*log(xx). I also moved everything on one line and used the return statement.

The testing code:

```
library(testthat)

test_that("my.dgamma.log matches dgamma() with log = TRUE", {
  expect_equal(my.dgamma.log(seq(0.2, 5, by = 0.2), shape = 0.2, rate = 5),
               dgamma(seq(0.2, 5, by = 0.2), shape = 0.2, rate = 5, log = TRUE))
  expect_equal(my.dgamma.log (seq(0.4, 10, by=0.4), shape=0.2, rate=5),
              dgamma(seq(0.4, 10, by=0.4), shape=0.2, rate=5, log=TRUE))
  expect_equal(my.dgamma.log(seq(1, 10, by=0.2), shape=2, rate=6),
```

```
              dgamma(seq(1, 10, by=0.2), shape=2, rate=6, log=TRUE))
})
```

```
## Test passed
```

The changes that I made 4. `my.dnorm.log()` generates the log of the probability density of the normal distribution. Note that this is exactly equal to the function `dnorm(..., log=TRUE)` so use this to verify your answers.
The function to be debugged: (Hint: Tests can pass with 1 change)

```
my.dnorm.log <- function(xx, mean, sd) {
  return(-log(sqrt(2*pi)*sd) - (xx-mean)^2/(2*sd^2) )
}
```

I followed the formula of a pdf of a normal distribution and used log on it to generate the right formula. Then, I corrected the code based on the equation that I attained. I changed the formulatp squre sd in the second part and modified it a little to use sqrt() function to get the square root of 2pi. **The testing code:**

```
test_that("my.dnorm.log matches dnorm() with log = TRUE", {
  expect_equal(my.dnorm.log(seq(-3, 3, by=0.2), mean=0, sd=1),
               dnorm(seq(-3, 3, by=0.2), mean=0, sd=1, log=TRUE))
  expect_equal(my.dnorm.log(seq(-3, 3, by=0.2), mean=-1, sd=2),
               dnorm(seq(-3, 3, by=0.2), mean=-1, sd=2, log=TRUE))
  expect_equal(my.dnorm.log(seq(-3, 3, by=0.2), mean=-2, sd=5),
               dnorm(seq(-3, 3, by=0.2), mean=-2, sd=5, log=TRUE))
})
```

```
## Test passed
```

5. The Fibonacci sequence [https://en.wikipedia.org/wiki/Fibonacci_number] is better defined dynamically, and one attempt at this is in `fibonacci()`. This function is supposed to generate the nth number in the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ..., which begins with 1, 1, and where every number after this is the sum of the previous two. (Hint: Tests can pass with 2 changes)

```
fibonacci = function(n) {
  if (n == 1 || n == 2) {
    return(1)
  }
  my.fib = c(1,1)
  for (i in 2:(n-1)){
    my.fib[i+1] = my.fib[i] + my.fib[i-1]
  }
  return(my.fib[n])
}
```

```
test_that("fibonacci functions gives correct output",{
 expect_equal(fibonacci(1), 1)
 expect_equal(fibonacci(2), 1)
 expect_equal(fibonacci(3), 2)
 expect_equal(fibonacci(5), 5)
 expect_equal(fibonacci(9), 34)
})
```

```
## Test passed
```

I added an if statement to deal with the edge cases. ## Testing

Now for a bit of unit testing. Download and install the `testthat` package if you haven't already. We'll use the `test_that()` function. It works as follows. Each call we make to `test_that()` has two arguments: the

first is message that describes what we are testing, and the second is a block of code that evaluates to TRUE or FALSE. Typically the block of code will use an `expect_*()` function, as seen in the debugging examples, in the last line. The structure is thus:

```
test_that("Message specifying what we're testing", {
  some code can be here
  some more code can be here
  expect_*(more code goes here)
})
```

If the output of your code is TRUE (the test passes), then the call to `test_that()` will show a message the test passed; if the output of your code is FALSE (the test fails), then we'll get an error message signifying this.

6. Finish writing the following unit tests for `add.up.inv.powers()` as indicated by the message.

```
library(testthat)
```

   a. Test the output for a single case, `n=3`, by writing manually the formula from 2 for `n=3` inside the test code.

```
test_that("add.up.inv.powers() works for n=3", {
    ## Put your test code here
  expected_output = 1 + sqrt(2) + 3^(1/3)
  actual_output = add.up.inv.powers(3)
  expect_equal(actual_output, expected_output)

})
```

   b. Test the output errors for input `n="c"`.

```
test_that("add.up.inv.powers() fails for non-integer n", {
    ## Put your test code here

  expect_warning(expect_error(add.up.inv.powers("c")))
})
```

```
## Called from: add.up.inv.powers("c")
## debug at <text>#5: for (i in 1:n) {
##     x = x + i^(1/i)
##     if (verbose)
##         roman.cat(i)
## }
## Test passed
```