HW 4 Random Number Generators and Latex

STAT 5400

Due: Sep 27, 2024 9:30 AM

Submit your solutions as an .Rmd file and accompanying .pdf file. Include all the **relevant** R code and output.

Always comment on your result whenever it is necessary.

Problems

1. Uniform random number generators in R

- Explore the source code of the R function `runif.R` by yourself. (You don't need to submit anything about this sub-question.)

https://github.com/wch/r-source

The architecture of the R function `runif` has several layers: (1) the R API: `runif.R` (2) the C code: `r-source/src/nmath/runif.c` which calls `rand_unif()` (3) the algorithm layer: `r-source/src/main/RNG.c` which implements several algorithms and Mersenne-Twister is the default algorithm.

- The Mersenne-Twister is a bit complicated. Let us work on a simpler algorithm, Linear Congruential Generator (LCG) Method. The LCG method is used to generate a sequence of pseudo-random numbers, and it is one of the oldest and simplest methods for generating pseudo-random numbers.

A tutorial of LCG is seen as in Section 9.3.2 of

https://homepage.stat.uiowa.edu/~luke/classes/STAT7400-2023/_book/simulation.html#uniform-random-numbers

The LCG is defined by the linear congruential relation:

$$X_{n+1} = (aX_n + c) \mod m$$

Where:

- X is the sequence of pseudo-random numbers
- a is the multiplier
- c is the increment
- m is the modulus
- X_0 (the seed) is the starting value of the sequence

Each number in the sequence is generated using the previous number. The results, which are modulo m, are normalized by dividing by m to fall within the interval [0,1).

Below is the pseudo Code for LCG

```
Algorithm LinearCongruentialGenerator
Input:
  n: the number of random numbers to generate
  a: the multiplier
  c: the increment
  m: the modulus
  seed: the starting value (seed) of the sequence
Output:
  A list of n pseudo-random numbers between 0 and 1.

Procedure:
  1. Initialize state to seed
  2. Initialize an empty list, random_numbers
  3. For i from 1 to n do
      a. state <- (a * state + c) mod m
      b. Append state/m to random_numbers
  4. End For
  5. Return random_numbers
```

- Based on the pseudo-code provided above, write an R function, named `LCG`, to generate `n` pseudo-random numbers between 0 and 1 using the Linear Congruential Generator method. The function should take five parameters: `n`, `a`, `c`, `m`, and `seed`.

```r
LCG <- function(n, a, c, m, seed) {
  #n <- #the number of random numbers to generate
  #a <-#is the multipler
  #c <- #the increment
  #m <- #the modolus
  state <- seed
  random_numbers <- numeric(n)
  for (i in 1:n) {
    state <- (a * state + c) %% m
    random_numbers[i] <- state /m
  }
  return(random_numbers)





}

n <- 10
a <- 1103515245
c <- 12345
m <- 2^31
seed <- 5400
```

```
random_numbers <- LCG(n, a, c, m, seed)
print(random_numbers)

##  [1] 0.8673853 0.8263298 0.8333236 0.3097559 0.4540139 0.2848278 0.8476917
##  [8] 0.5272413 0.6589490 0.5163056
```

Example Here's a set of parameters you can use to test your function:

a=1103515245 (Multiplier) c=12345 (Increment) m=2^31(Modulus)

```
random_numbers <- LCG(n = 10, a = 1103515245, c = 12345, m = 2^31, seed = 540
0)
print(random_numbers)
```

which basically functions the same to

```
set.seed(5400)
runif(10)
```

2. Q-Q plot

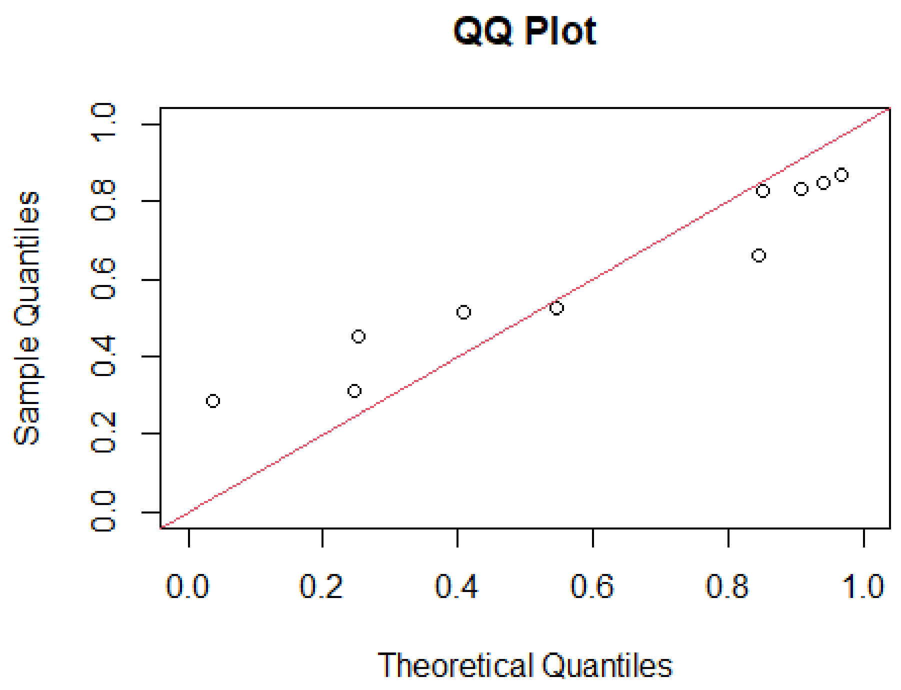
Call your `LCG` function to generate 10 pseudo-random numbers from a Uniform (0, 1) distribution. Draw a Q-Q plot to check if the numbers follow the Uniform (0, 1) distribution.

We will talk about the Q-Q plot in Monday's lecture. Several R packages provide implementations for Q-Q plots with a uniform distribution. You may explore this before the class.

```
set.seed(5400)
uniform_sample <- runif(n)

# Create a QQ plot to compare the LCG numbers with the uniform distribution
plot(
  sort(uniform_sample),           # Sorted sample from uniform distribution
  sort(random_numbers),           # Sorted random numbers from LCG
  xlim = c(0, 1),                 # x-axis limit
  ylim = c(0, 1),                 # y-axis limit
  xlab = "Theoretical Quantiles", # x-axis label
  ylab = "Sample Quantiles",       # y-axis label
  main = "QQ Plot"                # Title of the plot
)

# Add a reference line for y = x (perfect fit line)
abline(0, 1, col = 2)
```



3. LaTeX & Tikz library I

- Read the document latex.pdf in Tech Guide on the ICON site. Try to generate latex.pdf using the tex code.
- Use LaTeX to produce a pdf with the following table.

- Generate a very simple plot in R and save it as a pdf or eps. Include it in your tex code and thus in the pdf document.

- Instead of saving the plot as a pdf or eps, try the following R code before the plot is produced by R.

```r
library(tikzDevice)
tikz('myplot.tex', width=5, height=5)
plot(...)
```

- Add `\usepackage{tikz}` in your tex code (before `\begin{document}`) and include the tikz-produced plot using `\input{myplot.tex}`

No need to submit the pdf but attach all of your tex code in your R Markdown file.

Below is a table that displaus the probabilty density function.

4. LaTeX & Tikz library II

- Use the tikz package in latex to reproduce the following plot.

Below are some code you may begin with, although you do not have to use them.

```
\tikzstyle{block} = [rectangle, draw, fill=blue!20,
    text width=10em, text centered, rounded corners, minimum height=5em]
\tikzstyle{line} = [draw, very thick, color=black!80, -latex']
\tikzstyle{cloud} = [draw, ellipse,fill=red!20, node distance=2.5cm,
    minimum height=5em, text width=7em]


\begin{tikzpicture}[scale=0.5, auto]
    % Place nodes
    \node [block] (pop) {population};
    \node [block, right of=pop, node distance=7cm] (rs) {data (sample) \\ $\m
athbf{X} = (X_1, X_2, \ldots, X_n)$};
    <...>
    <...>
    % % Draw edges
    \path [line] (pop) -- (rs) node[pos=0.5, above, align=left] {sampling};
    <...>
    <...>
    <...>
\end{tikzpicture}
```

Diagram of Population, Data, and Parameters

5. (Coding practice) Regrouping problem

Suppose 30 numbers are grouped into six groups with group sizes 5, 5, 5, 5, 5, 5:

```
oldgroup <- split(1:30, rep(1:6, rep(5, 6)))
```

Write an R function `regroup`` to randomly form these numbers in new groups. It is required that (1) the group sizes, i.e., (5, 5, 5, 5, 5, 5), are unchanged and (2) any old group partners are not in the same group again.

```
> old_grp <- split(1:30, rep(1:6, rep(5, 6))))
>
> regroup <- function(grp){
<...>
+ }
>
>
> regroup(old_grp)
[[1]]
[1]  2 10 12 17 21
```

```
[[2]]
[1] 28  3  6 11 16

[[3]]
[1] 23 26  1  8 14

[[4]]
[1] 18 25 27  5  9

[[5]]
[1] 15 19 24 30  4

[[6]]
[1]  7 13 20 22 29
```

```
regroup <- function(group) {
  group.number <- length(group)          # Number of groups
  group.size <- length(group[[1]])       # Size of each group (assuming all a
re the same size)

  # Flatten the input list into a vector
  group.vector <- unlist(group, use.names = FALSE)

  # Initialize an empty vector to hold the new arrangement
  new_group.vector <- vector("numeric", length = group.number * group.size)

  # Rearrange elements into the new group vector
  for (k_index in 1:group.size) {
    for (i_index in 1:group.number) {
      # Fill the new group vector
      new_group.vector[(i_index - 1) * group.size + k_index] <- group.vector[
(k_index - 1) * group.number + i_index]
    }
  }

  # Split the new vector into a list of groups
  new_group <- split(new_group.vector, rep(1:group.number, each = group.size)
)

  return(new_group)
}

new_group <- regroup(oldgroup)
print(new_group)
```

```
## $`1`
## [1]  1  7 13 19 25
##
## $`2`
## [1]  2  8 14 20 26
##
## $`3`
## [1]  3  9 15 21 27
##
## $`4`
## [1]  4 10 16 22 28
##
## $`5`
## [1]  5 11 17 23 29
##
## $`6`
## [1]  6 12 18 24 30
```