STATS 506 Problem Set #2

Haiming Li

Dice Game

a. Here are different implementations of the function:

```
#' simulation version 1: loop implementation
#' @param n number of plays to make
#' @param seed seed to control random
#' @return final payoff
play_dice1 <- function(n, seed=NULL) {
  # input sanitation
  if (n < 1) {
    return(0)
  }

  res <- -2 * n
  set.seed(seed)
  rolls <- sample(1:6, n, replace=TRUE)
  for (roll in rolls) {
    if (roll == 3 | roll == 5) {
      res <- res + 2 * roll
    }
  }
  return(res)
}

#' simulation version 2: vectorized implementation
#' @param n number of plays to make
#' @param seed seed to control random
#' @return final payoff
play_dice2 <- function(n, seed=NULL) {
  # input sanitation
  if (n < 1) {
```

```
    return(0)
  }

  set.seed(seed)
  rolls <- sample(1:6, n, replace=TRUE)
  # replace payoff of all loss with 0
  rolls[which(!(rolls == 3 | rolls == 5))] <- 0
  return(2*sum(rolls) - 2*n)
}

#' simulation version 3: table implementation
#' @param n number of plays to make
#' @param seed seed to control random
#' @return final payoff
play_dice3 <- function(n, seed=NULL) {
  # input sanitation
  if (n < 1) {
    return(0)
  }

  # construct table with factor (predetermined levels)
  set.seed(seed)
  rolls <- table(factor(sample(1:6, n, replace=TRUE), 1:6))
  # calculate final payoff & remove name of vector
  res <- 2*(rolls[3]*3 + rolls[5]*5) - 2*n
  names(res) <- NULL
  return(res)
}

#' simulation version 4: table implementation
#' @param n number of plays to make
#' @param seed seed to control random
#' @return final payoff
play_dice4 <- function(n, seed=NULL) {
  # input sanitation
  if (n < 1) {
    return(0)
  }

  set.seed(seed)
  rolls <- sample(1:6, n, replace=TRUE)
  # apply a function that return the winning value of a given roll
```

```
  res <- vapply(rolls, function(roll) {
    if (roll == 3 | roll == 5) {
      return(2 * roll)
    }
    return(0)
  }, numeric(1))
  return(sum(res) - 2*n)
}
```

b. Here are some demonstrations:

```
cat("Functions with input n=3\n")
cat("play_dice1:", play_dice1(3), '\n')
cat("play_dice2:", play_dice2(3), '\n')
cat("play_dice3:", play_dice3(3), '\n')
cat("play_dice4:", play_dice4(3), '\n\n')
cat("Functions with input n=3000\n")
cat("play_dice1:", play_dice1(3000), '\n')
cat("play_dice2:", play_dice2(3000), '\n')
cat("play_dice3:", play_dice3(3000), '\n')
cat("play_dice4:", play_dice4(3000), '\n')
```

```
Functions with input n=3
play_dice1: 0
play_dice2: 4
play_dice3: 10
play_dice4: 4

Functions with input n=3000
play_dice1: 1870
play_dice2: 1848
play_dice3: 1650
play_dice4: 2032
```

c. Here are some demonstrations with seed 123:

```
cat("Functions with input n=3\n")
cat("play_dice1:", play_dice1(3, 123), '\n')
cat("play_dice2:", play_dice2(3, 123), '\n')
cat("play_dice3:", play_dice3(3, 123), '\n')
cat("play_dice4:", play_dice4(3, 123), '\n\n')
cat("Functions with input n=3000\n")
```

```r
cat("play_dice1:", play_dice1(3000, 123), '\n')
cat("play_dice2:", play_dice2(3000, 123), '\n')
cat("play_dice3:", play_dice3(3000, 123), '\n')
cat("play_dice4:", play_dice4(3000, 123), '\n')
```

```
Functions with input n=3
play_dice1: 6
play_dice2: 6
play_dice3: 6
play_dice4: 6

Functions with input n=3000
play_dice1: 2174
play_dice2: 2174
play_dice3: 2174
play_dice4: 2174
```

d. Here are speed comparisons. It seems that the implementation with apply is the slowest, the explicit loop implementation is the second slowest. This make sense because apply is loop hiding, and by passing in a function it creates extra overhead compared to explicit loop. The vectorized implementation is the fastest, and the table implementation is the second fastest. This also makes sense, it both of them leverage the speed of C, while the vectorized implementation have less part that need to run in R.

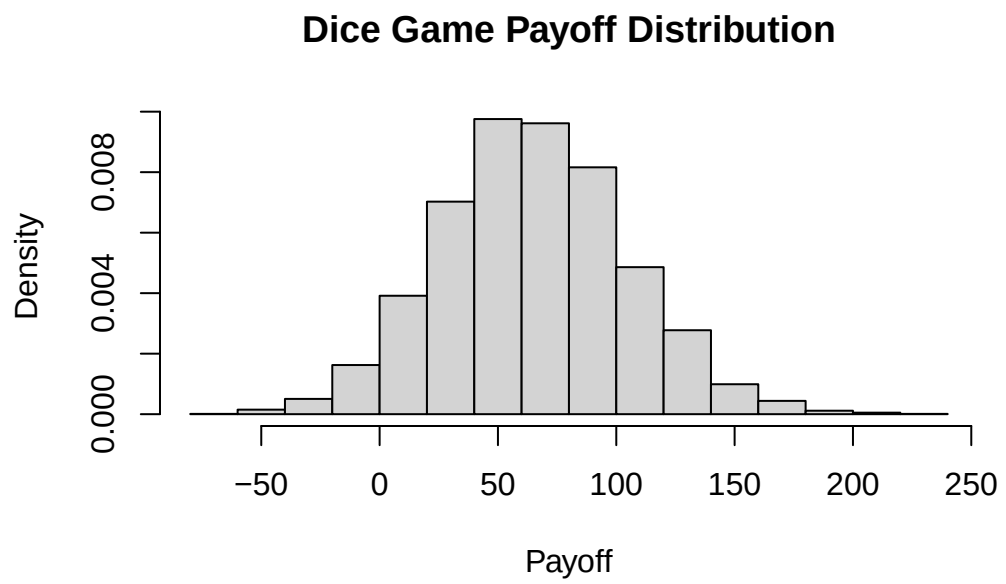
```r
library(microbenchmark)

microbenchmark(
  play_dice1 = play_dice1(1000, 123),
  play_dice2 = play_dice2(1000, 123),
  play_dice3 = play_dice3(1000, 123),
  play_dice4 = play_dice4(1000, 123)
)

microbenchmark(
  play_dice1 = play_dice1(100000, 123),
  play_dice2 = play_dice2(100000, 123),
  play_dice3 = play_dice3(100000, 123),
  play_dice4 = play_dice4(100000, 123)
)
```

```
Unit: microseconds
```

```
       expr     min       lq      mean   median       uq      max neval
 play_dice1  86.551  88.5395  94.27745  91.5325  97.006  143.787   100
 play_dice2  32.841  34.6860  38.53549  35.8750  39.770   68.388   100
 play_dice3  74.292  77.7975  84.59202  79.3350  88.068  152.151   100
 play_dice4 293.232 299.8125 341.16059 308.0740 329.271 1664.108   100
Unit: milliseconds
       expr       min        lq      mean    median        uq       max neval
 play_dice1  8.466008  8.649627  9.101248  8.878325  9.492422 12.566746   100
 play_dice2  3.212555  3.294596  3.370480  3.350213  3.435759  3.624974   100
 play_dice3  5.195684  5.391705  5.493877  5.482131  5.578542  5.796170   100
 play_dice4 30.424460 31.421191 32.325398 31.924753 32.466014 51.215519   100
```

e. It looks like the game is not fair, as the histogram is not centered around 0. This makes sense, as the expected payoff for each toss is $\frac{6+10}{6} - 2 = \frac{2}{3}$. The player is expected to gain.

```
res <- c()
for (i in 1:10000) {
  res <- append(res, play_dice2(100))
}
hist(res, main='Dice Game Payoff Distribution', xlab='Payoff', freq=FALSE)
```



Linear Regression