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
```

```r
  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:

```r
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: 4
play_dice2: 20
play_dice3: 16
play_dice4: 0

Functions with input n=3000
play_dice1: 1972
play_dice2: 2286
play_dice3: 1710
play_dice4: 1824
```

c. Here are some demonstrations with seed 123:

```r
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.

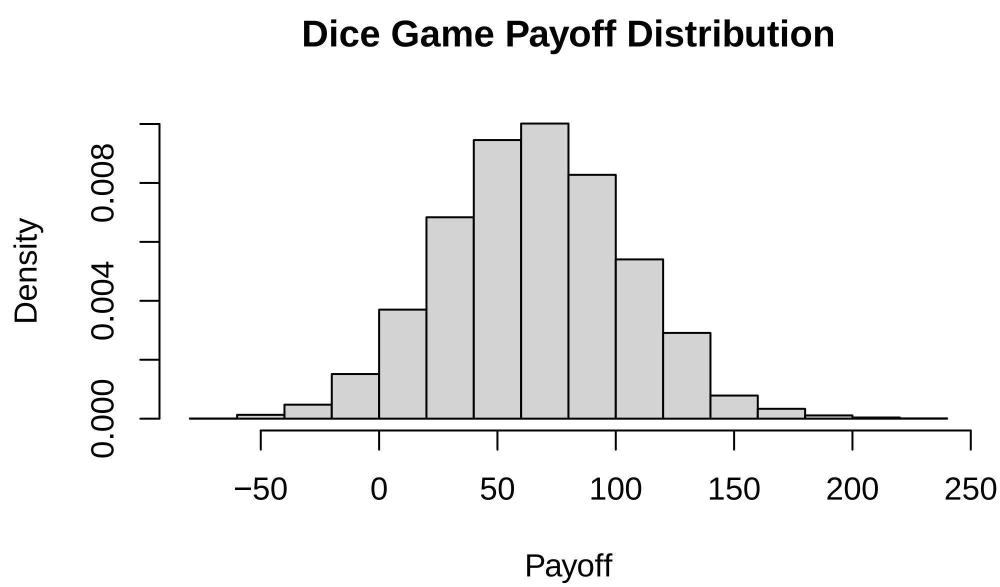
```r
library(microbenchmark)

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  86.428  87.8425  91.63500  88.7855  92.783  129.027   100
 play_dice2  32.882  34.1120  36.20300  35.1370  37.720   62.566   100
 play_dice3  69.454  72.6315  77.51255  74.3330  78.023  146.821   100
 play_dice4 291.387 295.4255 326.97213 301.1655 314.593 1339.265   100
Unit: milliseconds
        expr       min        lq      mean    median        uq       max neval
 play_dice1  8.358055  8.584170  9.010747  8.772401  9.307595 12.865595   100
 play_dice2  3.175409  3.267905  3.334951  3.315629  3.378113  3.688155   100
 play_dice3  4.769038  4.938716  5.041574  5.014587  5.125328  5.510482   100
 play_dice4 30.029302 30.911520 31.836378 31.298559 32.023071 49.335382   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.

```r
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

a. Here's the dataset with shortened column name.

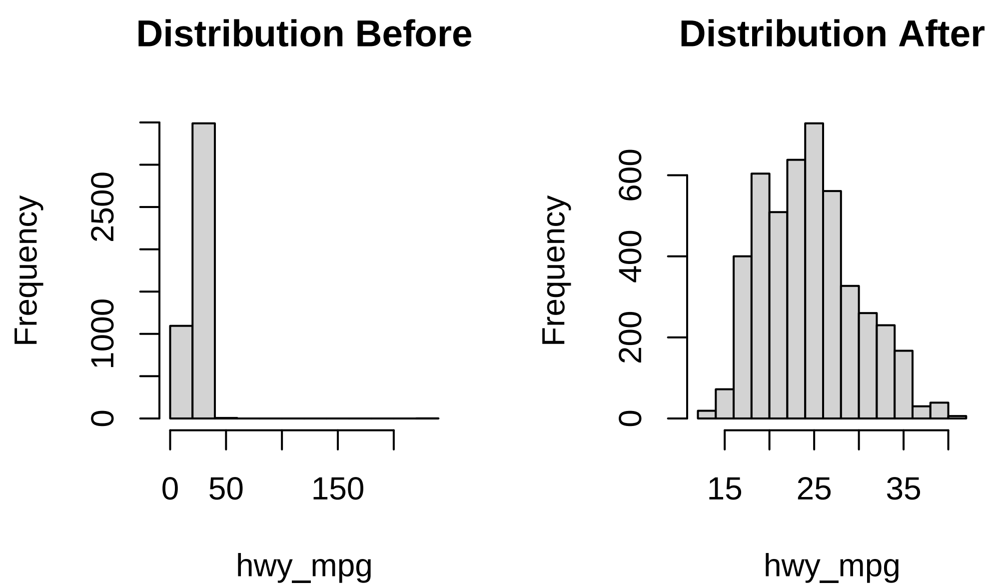
```
cars <- read.csv('cars.csv')
names(cars) <- c(
  "height", "length", "width", "driveline", "engine_type", "hybrid",
  "gears_cnt", "transmission", "city_mpg", "fuel_type", "hwy_mpg",
  "class", "id", "make", "model", "year", "horsepower", "torque"
)
```

b. Here's the filtered dataset.

```
cars <- subset(cars, fuel_type == 'Gasoline')
```

c. There's an extreme value in highway mpg. Removing it can be helpful to linear regression.

```
par(mfrow= c(1,2))
hist(cars$hwy_mpg, main='Distribution Before', xlab='hwy_mpg')
cars <- cars[-which.max(cars$hwy_mpg),]
hist(cars$hwy_mpg, main='Distribution After', xlab='hwy_mpg')
```



d. It seems that, while holding all else constant, a unit increase in torque would corresponds to 0.051748 decrease in highway mpg on average.

```
cars$year <- as.factor(cars$year)
model_fit <- lm(hwy_mpg ~ torque + horsepower + height +
                  length + width + year,
                data = cars)
summary(model_fit)
```

```
Call:
lm(formula = hwy_mpg ~ torque + horsepower + height + length +
    width + year, data = cars)

Residuals:
     Min       1Q   Median       3Q      Max
-10.9695  -2.4981  -0.3671   2.4164  19.8079

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 32.4107371  0.5486291  59.076   <2e-16 ***
torque      -0.0517480  0.0016727 -30.937   <2e-16 ***
horsepower   0.0171254  0.0017290   9.905   <2e-16 ***
height       0.0104613  0.0008555  12.228   <2e-16 ***
length       0.0011433  0.0006710   1.704   0.0885 .
width       -0.0008147  0.0006867  -1.186   0.2356
year2010    -0.4497677  0.5138721  -0.875   0.3815
year2011     0.0709912  0.5130245   0.138   0.8899
year2012     1.2925749  0.5170481   2.500   0.0125 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.494 on 4581 degrees of freedom
Multiple R-squared:  0.5626,    Adjusted R-squared:  0.5618
F-statistic: 736.5 on 8 and 4581 DF,  p-value: < 2.2e-16
```

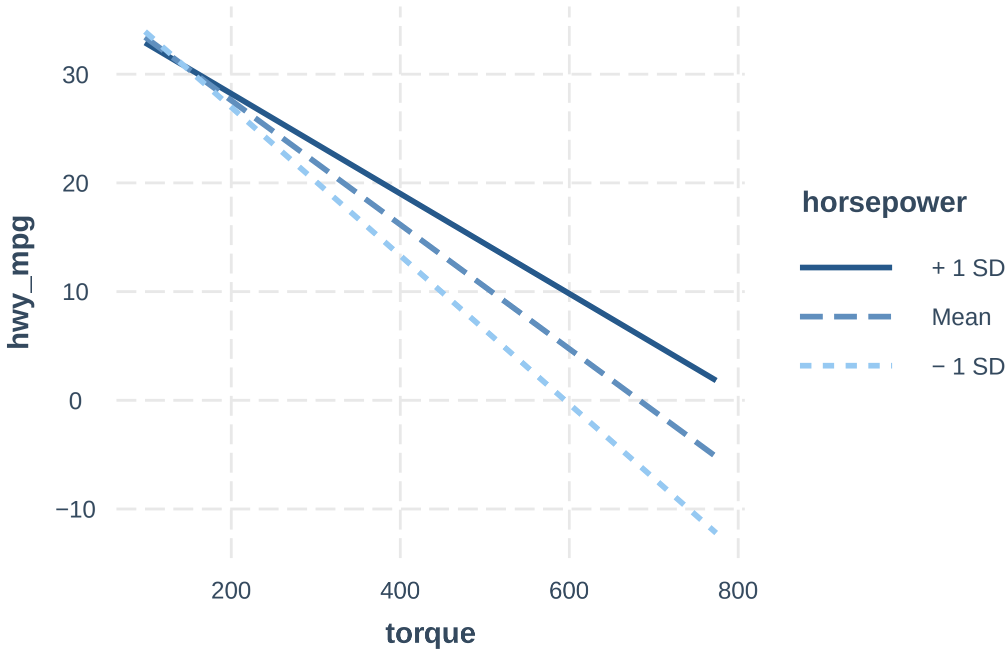
e. As shown, year 2011 have most data. Thus, I will use 2011 in the interaction plot.

```
table(cars$year)
```

```
2009 2010 2011 2012
  48 1633 1793 1116
```

Here's the interaction plot with year 2011.

```
library(interactions)
model_fit <- lm(hwy_mpg ~ torque * horsepower + height +
                  length + width + year,
                data = cars)
interact_plot(model_fit, pred = torque, modx = horsepower,
              at = list(year = factor(2011, levels(cars$year))))
```



f. For OLS, we have $\hat{\beta} = (X^T X)^{-1} X^T Y$.

```
X <- model.matrix(hwy_mpg ~ torque * horsepower + height +
                    length + width + year,
                  data = cars)
Y <- cars$hwy_mpg
solve(t(X) %*% X) %*% t(X) %*% Y
```

```
                       [,1]
(Intercept)     42.471008512
torque          -0.087656308
horsepower      -0.016438826
height           0.007060713
```

```
length             0.001189191
width             -0.001665871
year2010          -0.560374864
year2011          -0.029665866
year2012           1.184873242
torque:horsepower  0.000114224
```