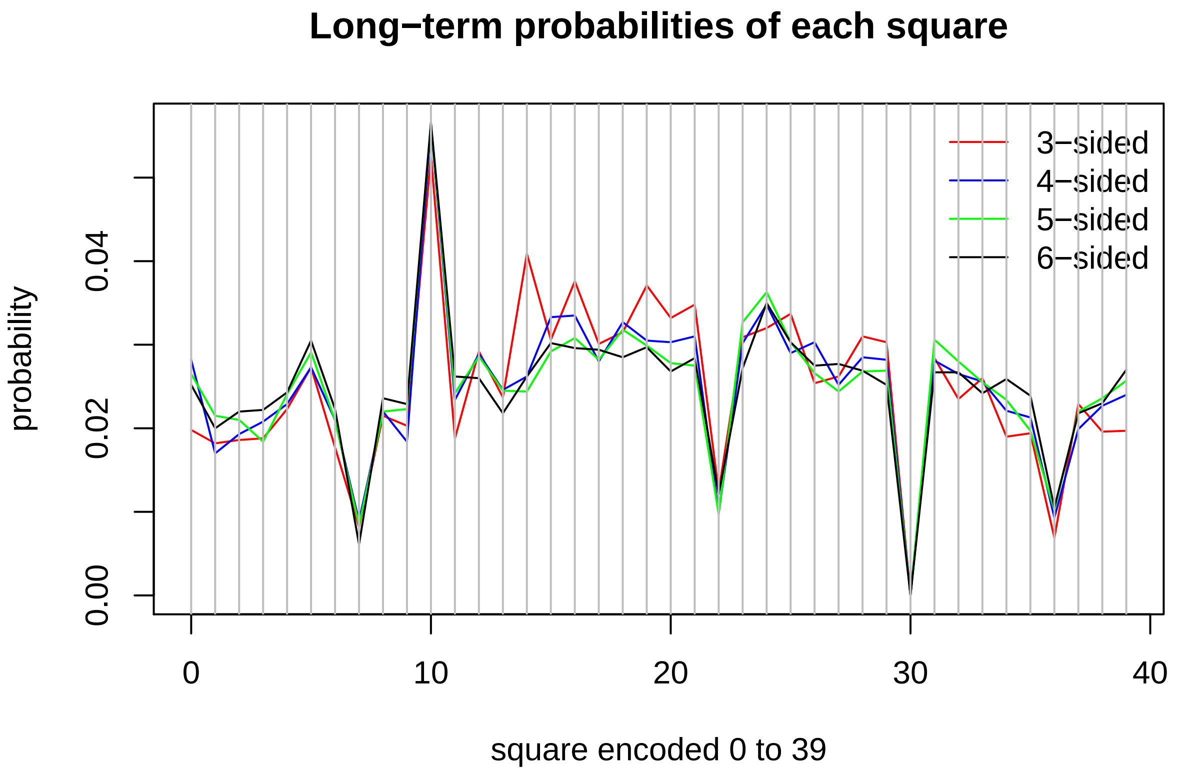
Monopoly Simulation

Ho Kwong Esmond, Chu

11/20/2017

1: simulate_monoploy function

2: estimate_monopoly function



We take n=10,000 to estimate the long-term probabilities of ending a turn on each Monopoly square (encoded as numbers from 0 to 39, 0 refers to the first "GO" square). For using a 6-sided dice, the 3 most likely squares to end a turn are 10, 24 and 5; for a 5-sided dice, the 3 most likely squares to end a turn are 10, 24 and 23; for a 4-sided dice, the 3 most likely squares to end a turn are 10, 24 and 16; and for a 3-sided dice, the 3 most likely squares to end a turn are 10, 14 and 16.

From the graph, for all 3-sided, 4-sided, 5-sided and 6-sided dice, landing on the square JAIL (encoded as 10) has the largest probability, this is because other than normal rolling, there are other conditions that make the turn lands on this square, including, landing on G2J (encoded as 30, this is also why the probability of landing on this square is 0), $\frac{1}{16}$ chance will be landing to JAIL when landing on CC1, CC2 and CC3 (encoded as 2, 17 and 33 respectively) and another $\frac{1}{16}$ chance will be landing on JAIL when landing on CH1, CH2 and CH3 (endcoded as 7,22 and 36 rspectively). 7, 22 and 36 have relatively smaller probability than the others, these are the Chance squares. Chance squares has $\frac{10}{16}$ probability that will send the landing position of a turn to another square, hence, the lower the probability of the final landing location on these squares. Although, square 2, 17 and 33 are the Community Chests, which have a similar function as the Chance squares, the probability of landing on these square aren't as low. This is because there is only $\frac{2}{16}$ chance will be sent to other squares when landing on Community Chest, which is a way smaller chance than the Chance square.

Although from the graph, all 3-sided, 4-sided, 5-sided and 6-sided dices have similar overall trend, we notice within the range after passing JAIL (encoded as 10) to E1 (encoded 21), 3-sided and 4-sided dices, especially 3-sided dice have a larger probability to land on squares within this range. To illustrate this reason, we first have to state that JAIL (encoded as 10) has a overall higher probability to land on, which means there are lots of situations that the turns will take JAIL as a starting square. The maximum steps the player will go by 1 turns by using a 3-sided dice is 3+3=6 steps, while for 5-sided and 6-sided dice are 5+5=10 steps and 6+6=12 steps respectively, which the maximum cases of using 5-sided and 6-sided dice will exceed square 20, but, 3-sided dice will be still in this range. Hence, within the range between 10-20, or after JAIL (encoded as 10), 3-sided and 4-sided dices have a higher probability to land on squares in this range than 5-side and 6-sided dice. Worth notice, the area undergraph should be 1, so when the probability of 3-sided and 4-sided are higher in the above mentioned range than 5-sided and 6-sided dices, 3-sided and 4-sided should have a lower probability in other ranges, and indeed, the graph shows this fact.

3

We use k=1,000 simulations with n=10,000 turns each to estimate the standard error for the long-term probability of ending a turn in jail, which is encoded as 10. The standard error we estimate using the above setting is 0.00215198 .

4a

We use the non-parametric bootstrap with b=1,000 samples from a simulation of n=10,000 turns to estimate the standard error for the long-term probability of ending a turn in jail. The standard error we estimate from this non-parametric bootstrap is 0.05079492.

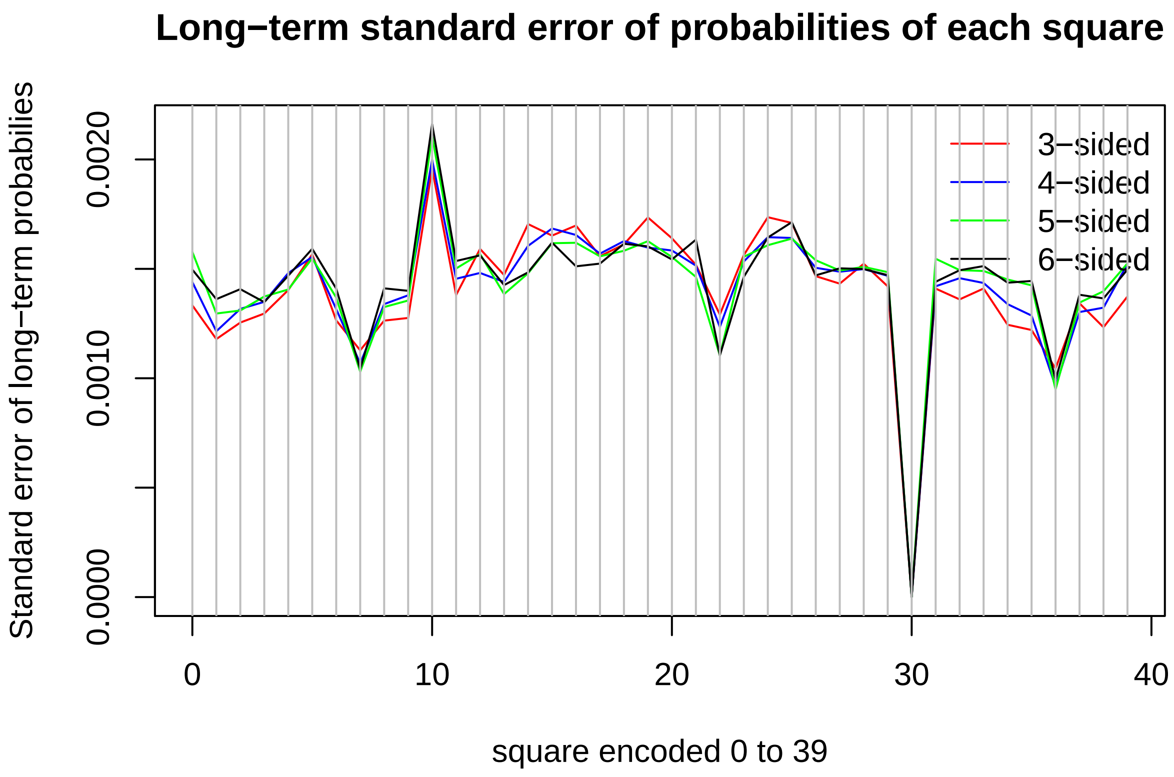
The simulation estimate gives a more accurant value, which 0.00215198 < 0.05079492, the estimate from bootstrap spread out a wider range of value and less accurate. The reason why is that, the result from the simulation is a real result from playing the game 10,000 turns and simulate this process for 1,000 times, this outcome is a real case simulation, while, bootstrap is based on the result from 1 game with 10,000 turns and keep doing resampling with replacement with the result in the set of outcome from just this game. So, the standard error for bootstrap is less accurate than the simulation.

4b

The bootstrap is much faster than simulation, as bootstrap is a process of random sampling with replacement, so it just keeps sampling from our sets of probability. Instead, the simulation process has to go through the **loop** (Please refers to question 1, simulate_monopoly() function) for 1,000 times and each turn consis of 10,000 turns. Thus, the bootstrap is much faster.

Although, the non-parametric bootstrap is obviously to be much faster than the simulation, we still gave them a speed test for a more scientific comparison. By using system.time(), we found that, the simluation took 306.221 seconds to run, while, the bootstrap took only 0.913 second. The bootstrap is undoubtedly faster than the simulation, yet, less accurant as explained above.

5



By reading the above graph that shows the standard error of the long-term probabilities of each square, it is similar to the earlier graph that presents the probability of landing on each square. That's the trend of the long-term standard error is the same as the trend of the long-term probability. For example, base on the long-term probability graph of each square, JAIL (encoded as 10) has a larger probability than other square, and from the stardard error graph, JAIL also has a larger value of standard error when comparing to others. Other than JAIL shows a peak, both graphs show a low on square 7, 22, 30 (the value for both graphs is 0 for square 30) and 36.

As standard error measures the spread, so this is natural enough to have a smaller value of standard error when the probability is small, which in our simulation, is the chance of landing on that square is small, vice versa.

6

The standard errors for the long-term probability estimates will tend to 0 as n increases. Since the estimated probability will get closer to the real probability.

R Appendix

```r
simulate_monopoly<-function(n,d){
  # n = number of turns by a player.
  # d = number of sides of each dice.
  community_chest = c(2,17,33)
  chance = c(7,22,36)
  location = 0
  double_counts = 0
  current_position = 0
  turn = 0

  cc_cards = rep(0, 16)
  cc_cards[sample(16,2)] = c(1,2)
  cc_cards = rep(cc_cards, ceiling(n/16))

  ch_cards = c(0,10,11,24,39,5,-1,-1,-2,-3,rep(-99,6))
  ch_cards = sample(ch_cards)
  ch_cards = rep(ch_cards, ceiling(n/16))

  while (turn<n) {
    turn = turn+1
    faces = sample(1:d,2,replace = TRUE)
    if (faces[1] == faces[2]) {
      double_counts = double_counts + 1
    } else {
      double_counts = 0
    }

    if (double_counts == 3) {
      current_position = 10
      double_counts = 0
    }

    steps = sum(faces[1],faces[2])
    current_position = current_position+steps
    if (current_position>39) {
      current_position=current_position%%40
    }
    if (current_position == 30) {
      current_position = 10
    }
    if (current_position %in% community_chest) {
      card1<-cc_cards[turn]
      if(card1 == 1){current_position == 0}
      if(card1 == 2){current_position == 10}
    }
    if (current_position %in% chance) {

      if (ch_cards[turn] >= 0){
        current_position = ch_cards[turn]
      } else if (ch_cards[turn] == -1) {
        if (current_position == 7){
          current_position = 15
        } else if (current_position == 22){
          current_position = 25
        } else {
          current_position = 5
        }
      } else if (ch_cards[turn] == -2) {
        if (current_position == 22){
          current_position = 28
        } else {
          current_position = 12
        }
      } else if (ch_cards[turn] == -3){
        current_position = current_position - 3
      }

    }

    location = c(location, current_position)
  }
  return(factor(location, 0:39))
}
estimate_monopoly = function (simulations) {
  return(prop.table(table(simulations)))
}

set.seed(95618)
all_probs = sapply(3:6, function(x) estimate_monopoly(simulate_monopoly(10000,x)))

plot(0:39, all_probs[,1], ylim=c(0, max(all_probs)),
     type='l', col="red", ylab="probability", main="Long-term probabilities of each square", xlab="squa
lines(0:39, all_probs[,2], col="blue")
lines(0:39, all_probs[,3], col="green")
lines(0:39, all_probs[,4], col="black")
legend("topright", c("3-sided", "4-sided", "5-sided", "6-sided"),
       col = c("red", "blue", "green", "black"), lty=1, bty='n')
abline(v=c(0:39), col="grey")

sort(all_probs[,4], decreasing = TRUE)[1:3]
sort(all_probs[,3], decreasing = TRUE)[1:3]
sort(all_probs[,2], decreasing = TRUE)[1:3]
sort(all_probs[,1], decreasing = TRUE)[1:3]
set.seed(94523)
jail_probs = sapply(1:1000, function(x) estimate_monopoly(simulate_monopoly(10000,6))[11])
sd(jail_probs) # 0.00215198
set.seed(95218)
one_time = simulate_monopoly(10000, 6)
bootstrap_samples = lapply(1:1000, function(x) sample(one_time, replace=TRUE))
bootstrap_probs = unlist(lapply(bootstrap_samples, function(x) estimate_monopoly(x)[11]))
bootstrap_probs[1] # 0.05079492
time_sim = function() {
set.seed(94523)
jail_probs = sapply(1:1000, function(x)
```

```r
estimate_monopoly(simulate_monopoly(10000, 6))[11])
sd(jail_probs) # 0.00215198
}

time_boot = function() {
set.seed(95218)
one_time = simulate_monopoly(10000, 6)
bootstrap_samples = lapply(1:1000, function(x)
sample(one_time, replace = TRUE))
bootstrap_probs = unlist(lapply(bootstrap_samples, function(x)
estimate_monopoly(x)[11]))
}

system.time(time_sim())
system.time(time_boot())

#system.time(time_sim())
  # user  system elapsed
  #247.137  58.945 306.221
#system.time(time_boot())
  #user  system elapsed
  #0.782   0.129   0.913
set.seed(94523)
squares_sd = list()
for (i in 3:6){
squares_probs = lapply(1:1000, function(x) estimate_monopoly(simulate_monopoly(10000,i)))
squares_sd[[i]] = apply(do.call("cbind", squares_probs), 1, sd)
}

all_sd = do.call("cbind", squares_sd)

plot(0:39, all_sd[,1], ylim=c(0, max(all_sd)),
     type='l', col="red", ylab="Standard error of long-term probabilies", main="Long-term standard erro
lines(0:39, all_sd[,2], col="blue")
lines(0:39, all_sd[,3], col="green")
lines(0:39, all_sd[,4], col="black")
legend("topright", c("3-sided", "4-sided", "5-sided", "6-sided"),
       col = c("red", "blue", "green", "black"), lty=1, bty='n')
abline(v=c(0:39), col="grey")
```