Jenny Goldsher
7/20/2023
ISYE 6644

Programmatically Modelling the Expected Value and Distribution of Turns in a Dice Game

Abstract

The present project aims to examine the expected value and distribution of turns of a dice game. Previous literature has outlined appropriate scenarios for the application of simulation and mathematical techniques to derive expected value and distributions of dice and card games. The implementation of simulation techniques allows for cross-comparison amongst different starting parameters and strategies of various games. Using R/R Studio, a simulation was developed to model and interpret the output of 20,000 iterations of the dice game. Monte Carlo methodologies, Pseudo-Random number generation, and random sampling were all integral in the creation of a reliable and replicable simulation. The simulation indicated that the distribution of turns per dice game is heavily right-tailed with a median of 14. Future research could investigate how different starting values and rule changes may affect the expected distribution and value of turns per game.

Background

Monte Carlo Simulation techniques offer a means of modeling distributions with random components at play. The ability to model scenarios that have finite components and random variables over a large sample enables the capacity to holistically describe the distribution on hand. Compared to other algorithms used to model closed-circuit simulations, Monte Carlo is generally less computationally heavy (Marin & Williamson, 2021). Monte Carlo techniques can also be integrated with other modelling frameworks, relevantly Markov Chain Monte Carlo. In scenarios where both properties of Markov Chains and Monte Carlo are relevant, each state-transition of the Markov chain have unique weighted components and random variables to be sample via Monte Carlo methodologies.

Casino games are a well-suited domain to apply Monte Carlo simulation techniques. Casino games offer appropriate properties for simulation modelling because both Dice Rolls and Decks of Cards can be represented as sets, and their associated randomness can be incorporated by using PRNs. Previous research has applied Monte Carlo techniques to the dice game “Craps” and outlined best-strategies based on their findings (Marin & Williamson, 2021). Researchers tested strategies of “biasing the dice” (favoring certain numbers more than others, assuming a situation with player control), and verified their results using Monte Carlo simulation (Marin & Williamson, 2021). In their study of Craps strategies, Marin and Williamson (2021) altered the setup of the game by reweighting dice probabilities. Subsequently they systematically applied Monte Carlo simulations to get the expected value of each scenario. This allowed them to make comparisons of the distributions and expected values across strategies with an acceptable degree of control and reliability (Marin & Williamson, 2021).

Another important component of game-simulations is the incorporation of state-to-state transitions, namely the application of Markov Chains (Prevault, 2018). At many points of

card/dice games probabilities change solely based on the previous state, making them applicable candidates for Markov Chain theory. Previous research investigated optimizing a blackjack AI by deriving expected value from a Markov chain (Yu, 2012). In this study, Markovian-dealer states were defined as all possible "face-up" dealer cards, and Markovian-player states as the possible sums of the first two cards (Yu, 2012). From there, state-to-state transition rates to were derived to compute expected value. Similarly, Wakin and Rozell (2004), investigated expected value of blackjack both in a single hand and throughout an entire "shoe" (multiple decks of cards) using Markovian techniques. Their research validated various player strategies of card counting, identifying that under certain conditions throughout the game there are different strategies a player can enact to maximize expected value (Rakin & Rozell, 2004).

Having a reliable Pseudo-Random number generator is key in modelling randomness within a simulation. Researchers investigated the merits of Linear Congruential generators (form: $X_{i+1} = aX_i + b \pmod{m}$, citing that they are popular in Monte Carlo simulations (Boyar, 1989). As we have seen in this course, carefully choosing values of *a,b* and *m* are key in developing independent and identically distributed (i.i.d.) uniform numbers. For a generator to be reliable, it must have a large, full-period and not have any biases (only evens/odds, have correlations – large follow large, small follow small). Many programming languages have built in reliable random number generators that can be used for modelling purposes.

Problem Definition

Previous research has shown that there is an extensive space for application of simulation techniques to reliability develop estimates for expected values and distributions of card and dice games. This project aims to apply Monte Carlo methods, Markovian principles, and the implementation of reliable PRN generators to derive those values for a dice game with the following rules:

"There are two players, A and B. At the beginning of the game, each starts with 4 coins, and there are 2 coins in the pot. A goes first, then B, then A,. . . . During a particular player's turn, the player tosses a 6-sided die. If the player rolls a:
• 1, then the player does nothing.
• 2: then the player takes all coins in the pot.
• 3: then the player takes half of the coins in the pot (rounded down).
• 4,5,6: then the player puts a coin in the pot.
A player loses (and the game is over) if they are unable to perform the task (i.e.,if they have 0 coins and need to place one in the pot). We define a cycle as A and then B completing their turns. The exception is if a player goes out; that is the final cycle (but it still counts as the last cycle)."

Method

To model the card game, I developed a simulation using R/R Studio. R has a built in random number generator *runif(),* which was used to generate random dice rolls. Additionally, R has built in ceiling (*ceiling()*) and floor (*floor()*) functions which were used to bucket *runif()* values to dice rolls and round-down the value of the pot respectively. Vectors (R's means for list storage) were

used to store the number of turns per game and value of the pot throughout each game. To both interpret and model the data, I used R's built in *hist(), boxplot(),* and *ggpplot()* functions (See Appendix A). The script executes 20,000 games while tracking the turns per game and value of the pot throughout the game (See Appendix A).

Mathematical Representations of Expected Value per dice roll

Before implementing the code simulation of the dice game, I looked at the mathematical representations of expected value. I derived the value of *E[Turn] = .25*Pot - .5* using the following p.m.f.:

Dice Value	***1***	***2***	***3***	***4***	***5***	***6***	
Likelihood	1/6	1/6	1/6	1/6	1/6	1/6	
Related Value	0	Pot	Floor(.5*Pot)	-1	-1	-1	
Weighted Value	0	1/6*Pot	1/12*Pot	-1/6	-1/6	-1/6	**Σ = .25*Pot - 1/2**

Given the derived p.m.f., we would expect dice roll #1 to yield *EV[Turn #1] = 0*:

$$E[\text{Turn \#1}] = .25 * (\text{Pot}) - ½$$
$$= .25*(2) - ½$$
$$= ½ - ½ = 0$$

Though the expected value of Turn #1 is 0, with the various outcomes of each dice roll, we would expect that the distribution will eventually change as the value of the pot fluctuates. Given the randomness of each dice roll, implementing a simulation would be suitable to model the distribution and expected value of the game.

Reliability of PRN Generation in R

As previously mentioned, R has a built in random number generation function *runif().* By default, *runif(x)* will return *x* i.i.d. values between 0 and 1 where *x* is any positive integer. To simulate each dice roll within the R, I coupled *runif()* and *the ceiling()* functions to get a discrete value between 1 and 6 (See Appendix A, line 117). To validate that this would in fact generate i.i.d. values between 1 and 6 over 20,000 iterations, I ran a chi-square test on the outputs as represented by the following table:

Dice Value	***1***	***2***	***3***	***4***	***5***	***6***	
Observations	3297	3336	3352	3350	3327	3338	
Expected Value	3333.3	3333.3	3333.3	3333.3	3333.3	3333.3	
(0i - E)^2/ E	(36.3)^2/3333.3 = .396	(2.67)^2/3333.3 = .002	(18.67)^2/3333.3 = .105	(16.67)^2/3333.3 = .084	(6.3)^2/3333.3 = .012	(4.67)^2/3333.3 = .007	**Σ = .606**

I used $X^2_{.01,1000} = 1133.579$ (the largest degree of freedom publicly available, with values ascending as df got larger so I was confident, if anything, I was undervaluing the critical value) as the critical value. The Chi-squared value of .606 is far less than the critical chi-squared value needed to reject the null hypothesis of uniformity. Thus, I found R's built in *runif()* function appropriate to use throughout the simulation.

Simulation: Expected Value and Distribution of Turns per Game

Upon 20,000 runs of the dice game, the simulation yielded a right-tailed distribution of turns per game as seen in Figure 1. Because the distribution was heavily skewed, I computed median and IQR as they were the most appropriate central tendency measures. The simulation had a median of 14 turns per game with an IQR of 14, a lower quartile of 8, and an upper quartile of 22.

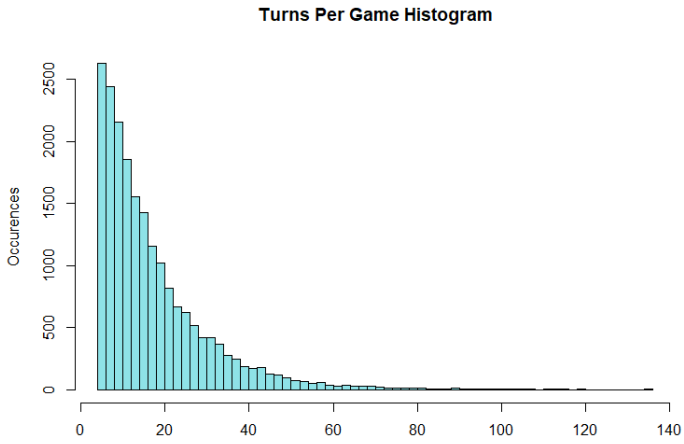

Figure 1 shows a histogram of turns per game over the 20,000 iterations of the simulation.

The derived expected value equation demonstrated that the EV of a given turn was highly dependent on the value of the pot at that dice roll. Since each iteration had a unique sequence of events, the value of the pot fluctuated accordingly on each roll. Figure 2 shows the pot values of 5 random games at each dice roll. Since each dice roll is random and independent, there are no general trends across runs of the simulation for pot value at each dice roll. Nonetheless, the variation between pot values across each simulation indicates that the expected value of turns per game would fluctuate accordingly.

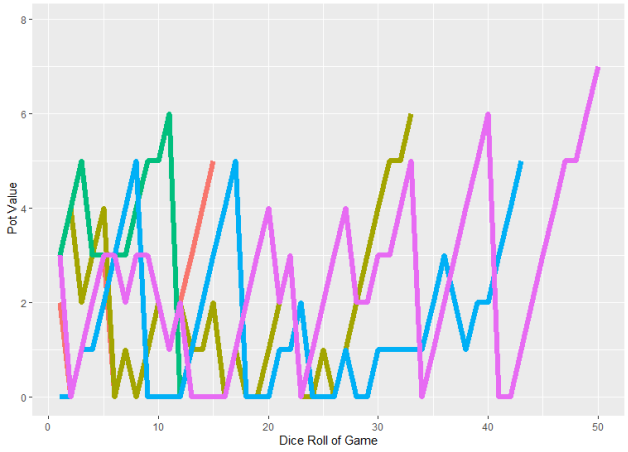

Figure 2 displays the value of the pot during five randomly selected runs of the simulation.

Conclusions

Simulation offers a reliable methodology for deriving the expected value and distribution of the aforementioned dice game. The simulation suggests that we should expect each dice game to last 14 turns, with the middle 50% of games having between 8 and 22 turns. Additionally, since the distribution of outcomes was heavily right-tailed, we may expect to see longer games at

a low rate. At each dice roll, the expected value of a player's turn is heavily dependent on the value of the pot at that point in time.

Similar to previous research (Marin & Williamson, 2021), Monte Carlo methodology was relevant in generating the distribution of turns in the dice game. In the present study, an extensive number of iterations of the game gave us a complete picture of the distribution and expected value of the game without having to enact complex mathematical modelling. Further research could examine how the distribution of turns in a game may change if the initial values of the pot and players were different. Related to the previously mentioned study conducted by Wakin and Rozell (2004), we could vary the strategy and probabilities across to make cross-comparisons. In addition, if the probability of a given dice roll was contingent on previous rolls, we could include Markovian principles in the simulation.

Programming allows for the dynamic integration of simulation techniques. Compared to complex mathematical models, it is far easier to alter the parameters of a simulation script to view how it affects related outputs. This project illuminated the value of the "built-in" functionality of various programming languages. Being able to rely on the random number functionality in R saved computational time and allowed for appropriate variation in the simulation. The benefits and applications of various programming languages (R, Arena, Python, etc.) in simulation modelling are extensive and make programming a robust option for future simulation models.

Boyar, J. (1989). *Inferring Sequences Produced by Psuedo-Random Number Generators*. Journal of ACM, *36*(1),129-141.

Marin, A., & Williamson, C. (2021). *Cheating at Craps*. ACM SIGMETRICS Performance Evaluation Review, *48*(4), 53–61.

Privault, N. (2018). *Understanding Markov Chains: Examples and Applications*. Springer.

Wakin, M., & Rozell, C. (2004). *A Markov Chain Analysis of Blackjack Strategy.* Rice University

Yu. J (2012). *Markovian Blackjack Simulator.* University of Massachusetts, https://inside.mines.edu/~mwakin/papers/mcbj.pdf

Appendix A

```r
#import relevant libraries for data manipulation and visualization
library(dplyr)
library(tidyr)
library(ggplot2)

#Initialize a vector to store number of turns
turn_vec= c()
#Initilialize dataframe for tracking pot values
pot_track = data.frame(
  iteration = c(0),
  turn = c(0),
  pot_val = c(0)
)
#run 20k simulations
for(i in 1:20000){
iters = c()
pot_values = c()
t_vals = c()
#initialize values for players and pot
a = 4
b = 4
pot = 2

#set value to zero for turns, true false for running
running = TRUE
turns = 0
#while loop for each iteration
while (running) {
  #generate a dice roll
  dice =ceiling(6 * runif(1))
  #update turns
  turns = turns + 1

  if(dice == 1){
    #nothing, proceed
  }else if(dice == 2){
    #player gets all the coins in the pot, use mod to see which player we are on
    if(turns %% 2 == 0){
      a = a + pot
    }else{
      b = b + pot
    }
    #Set pot to zero
    pot =0

  }else if(dice == 3){
```

```
    #player gets half the coins in the pot rounded down, use mod to see which player we are on
    if(turns %% 2 == 0){
      a = a + floor(.5 * pot)
    }else{
      b = b +  floor(.5 * pot)
    }
    #Set pot to zero
    pot = pot -  floor(.5 * pot)

  }else{
    #put one in pot, see if player is out
    if(turns %% 2 == 0){
      a = a - 1

      if(a < 0){
        running = FALSE
      }
    }else{
      b = b - 1

      if(b < 0){
        running = FALSE
      }
    }
    pot = pot + 1
  }

  #store values
  iters = c(iters, i)
  pot_values = c(pot_values, pot)
  t_vals = c(t_vals, turns)

}
#store values
this_pot_track = data.frame(
  iteration = iters,
  pot_val = pot_values,
  turn = t_vals
)
pot_track = rbind(pot_track, this_pot_track)
#account for one dice roll being half a turn, ceiling so if A was last still counts as a turn
turn_vec = c(turn_vec, ceiling(turns * .5))
}

pot_track = pot_track %>%
  mutate(iteration = as.factor(iteration))
#randomly select 5 iterations to view pot fluctuation
rand_iters = sample(1:20000, 5)
p_track =pot_track %>%
filter(iteration %in% rand_iters)
ggplot(p_track, aes(x = turn, y = pot_val, color = iteration)) +
  geom_line(size = 2.2)+
  ylab('Pot Value')+
  xlab('Dice Roll of Game')+
  ylim(0,8)+
  theme(legend.position="none")
pot_graphic

#create histogram of distribution of turns
h <- hist(turn_vec,breaks = 50, plot = FALSE)
p <- plot(h, ylab = 'Occurences', xlab = 'Turns', col = '#8ee2e7', main = 'Turns Per Game Histogram')

#look at central tendency measures for turns
median(turn_vec)
IQR(turn_vec)
mode(turn_vec)
boxplot(turn_vec)
fivenum(turn_vec)

#Check runif to see if uniform
tosses = ceiling(6 * runif(20000))
table(tosses)
```

Appendix A. Above shows the simulation code constructed in R studio that was used to model the dice game and generate the distribution of turns.