Mauricio Deguchi
UID: 405582562
CEE/MAE 20
November 9, 2021

Homework 6

1 Basketball Probability

1.1 Introduction

This script is intended to run a Monte Carlo simulation for 2 different strategies in the dying moments of a basketball game. In strategy a, we solely rely on 2-point shots and rebounds, while fouling our opponent to ensure they don't simply run down the clock. In strategy b, we rely on a 3 point buzzer-beater to force overtime and then we have even odds in overtime.

1.2 Models and Methods

For these simulations, we will have two main for loops, one for each strategy. The for loops will loop the inner parts for the specified amount of games. Prior to our loops, we establish our probabilities for each of the options outlines in the problem statement (with the exception of their 2-point shot as we were later clarified that this would be irrelevant).

For the over all problem, I decided it would be easiest to establish functions for each scenario depicted in the problem statement. This ended up breaking up into 6 different situations: (1) we are shooting a two pointer (our2Pointer), (2) we miss our 2 pointer and try to get the rebound (ourOffensiveRebound), (3) they have possession so we foul them and they take their freethrows (their possession), (4) they miss their second freethrow an have the opportunity to get the rebound (theirOffensiveReboundFree), (5) we are shooting a 3 pointer, (6) we go into overtime.

Once I declared all of the functions, I moved on to starting the 2 point strategy. Inside of our for loop, I started a while loop to check the time, ensuring that the game had not ended. As long as we had time we could use one of our functions to run a scenario. From there, it was a matter of establishing the conditions for when to use each function using if statements. These conditions were mostly outlined on the update on CCLE; the most ambiguous line is the following

```
if (possession == 0) && (time == 5)
    time = 0;
end
```

This statement inside the time loop checks if they have possession and we are down to the final five seconds (since we decrease in intervals of 5 seconds this is the last stop before we end the game). If they have possession with 5 seconds to go, the instant we foul them, the game will be over so there will be no time for freethrows. Using this allows us to have our freethrow loop be slightly cleaner without having to check for time.

1.3 Calculations and Results

Using the Monte Carlo approach, the script runs large amounts of trials. These trials loop through the game simulation, adding 1 to our win counter for each time we win a game. At the end of the simulation, we take out total number of wins for each strategy and divide it by the total number of simulations run for each strategy. This gives us the probability of winning (with some error due to the relatively small number of trials) for each strategy. I then reran the code using varying numbers of trials and switching the probabilities to get the following results:

	2 Point Win Rate	3 Point Win Rate
N = 300	5.667 %	18.667 %
N = 1000	4.100 %	18.300 %
N = 2000	4.200 %	17.400 %

Table 1.1 Win rate of each strategy for varying amounts of trials using the probabilities in part (a)

	2 Point Win Rate	3 Point Win Rate
N = 300	14.333 %	13.000 %
N = 1000	17.600 %	12.200 %
N = 2000	18.150 %	12.800 %

Table 1.2 Win rate of each strategy for varying amounts of trials using the probabilities in part (b)

Note that these probabilities are randomized, so if we were to rerun the same code again the results will look different. The higher our N, the closer we are to the true values, however our two data sets are distinct enough from which we can make a reasonably confident conclusion

1.4 Discussion

By using the data from table 1.1 and 1.2, we can see that the win rate for the 3 point buzzer-beater with the probabilities in part (a) is significantly higher than the win rate for the 2 point method. However, when we switch the probabilities to the ones described in part (b), we obtain very different results. For the largest trial, we have a difference of around 4.5% which is sufficiently convincing that with the second set of probabilities, we should use the 2 point strategy. Thus, the strategy to win the game varies depending on the probabilities of our shots and the opponent's shots.

2 Customized Probability Distribution

2.1 Introduction

This script will create a customized probability distribution, drawing samples from the function:

$$p(x) = \begin{cases} -\frac{2}{9}x + \frac{2}{3}, & if\ x \in [0,3] \\ 0 & otherwise \end{cases}$$

Equation 1.1 Probability density function provided in the problem statement

While this specific probability density may not have much significance, customized probability densities come up frequently in engineering and this problem will give insight as to how to create such customizations.

2.2 Models and Methods

The method for this problem is largely outlined in the problem statement. Our first step will be to manually integrate the probability density function to get our P(x). From there we will want to take the inverse of P(x) and lastly, we will generate and push a random set of numbers between [0,1] through our $P^{-1}(x)$ function to generate our random values for the probability density depicted in Equation 1.1.

2.3 Calculations and Results

So, let's begin by manually integrating our probability density function:

$$P(x) = \int_{-\infty}^{x} p(u)\, du$$

Equation 2.1 Integral depicted in the problem statement

$$P(x) = \int_{-\infty}^{x} -\frac{2}{9}u + \frac{2}{3}, du$$

Equation 2.2 Substitute in our p(u) from

$$P(x) = \int_{-\infty}^{0} 0\, du + \int_{0}^{x} -\frac{2}{9}u + \frac{2}{3}, du$$

Equation 2.3 Split up integral for the piecewise function

$$P(x) = -\frac{1}{9}x^2 + \frac{2}{3}x\,, for\ x \in [0,3]$$

Equation 2.4 Final evaluated integral

Next, I used an online calculator to solve for $P^{-1}(x)$:

$$x = P^{-1}(y) = \begin{cases} (a) & 3\sqrt{-y+1}+3 \\ (b) & -3\sqrt{-y+1}+3 \end{cases}$$

Equation 2.5 Our inverse function as a system of equations

We then implement equation 2.5 (a) into Matlab into the myRand function. We use equation (a) rather than equation (b) because we need to have the positive probability density from y = [0,1]. Once I had established the myRand function, I created a y array of randomly generated value between [0,1]. This y array could then be pushed through Equation 2.5 (a) to obtain our random values from the probability density depicted in Equation 1.1. From there, our new x array contains all of our randomly distributed values, so we can then create a histogram from our x array to see a rough outline of our probability density.

On the same figure, we also plot the line from Equation 1.1 to see how accurate our random distribution is to the actual function.

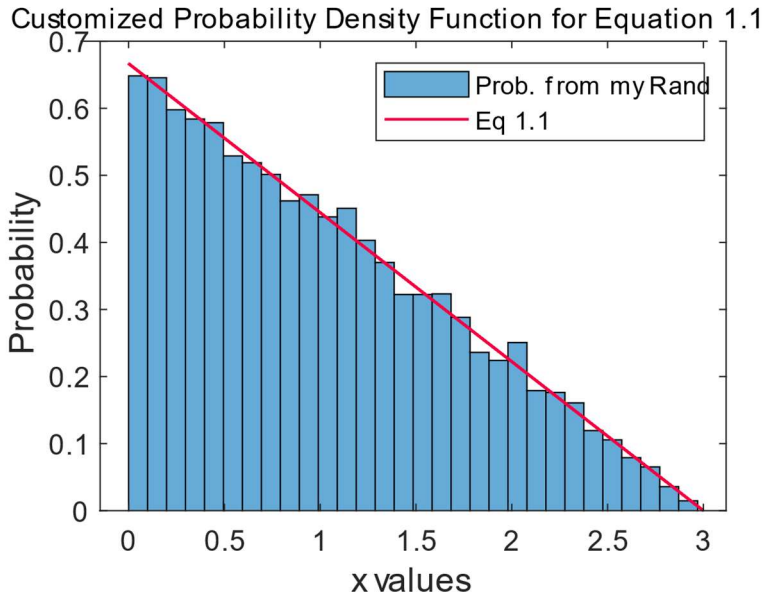

Figure 1 Probability Density Plot using a Monte Carlo simulation alongside the true probability

2.4 Discussion

Figure 1 displays our histogram and true probability density function. As expected, we see some deviation from the true probability curve. This is due to the nature of randomized simulations, however, if we were to increase our number of samples N we took, our histogram would be a more accurate approximation of the true value. Overall, this problem was very interesting and introduced me to probability distributions. While I had not been entirely with the reasoning behind some of the manual math we performed, I can now confidently say that I know how to create a probability distribution from a customized function which can be integrated.

3 Birthday Paradox

3.1 Introduction

The birthday paradox is the unprecedented probability of two people in a relatively small room having the same birthday. The high probability results from the exponentially increasing connections between people, since as each new person is added to the group, we must check if their birthday matches each individual in the group.

3.2 Models and Methods

The script first establishes some forefront values and arrays which will be necessary including the maximum number of people we want to examine in a room , the number of trials we want to test, and the arrays which will be reassigned values for matches later on.

Once this has been established, we begin our loops. Our outer most loop runs for the number of trials we want to run for the various groups of people we want to sample. We then move into an inner loop where we generate random integers between 1-365. The number of random numbers depends on the size of the group which is determined by the step we are in the loop. We can then sort the values into ascending order using the `sort` function. Sorting the numbers will allow us to check matching values easily by simply checking if consecutive numbers are equal. Inside of our previous loop, we begin another loop which will check all of the consecutive values to see if any of them match. If we get a single match, we exit stop checking the rest of the people for matches. This ensures that we get the probability of at least one match. Our loops run for each group size from 2 to our specified upper limit over the course of a large amount of specified trials. Over these trials, we store the matches into an array and sum the array over all of the trials to get the frequency of our matches for each group size.

3.3 Calculations and Results

For each trial, we have an array of length of the maximum group size and we store if we find a match in that trial for each group size as 1. Over the large number of trials, we sum up this array to find the total times we found at least one match in each group size. Then, we can divide this array by the total number of trials we conducted to get the probability of getting at least one match for each groups size in the array. This array is then divided by the total number of trials to get the probability of at least one match for each group size. So, our final array contains a list of probabilities for each group size corresponding to their position in the array.

Following that, we generate an array of integers to correspond with all of the group sizes we have. We then plot the group size array on the x-axis alongside our probabilities on the y-axis to generate the following plot.

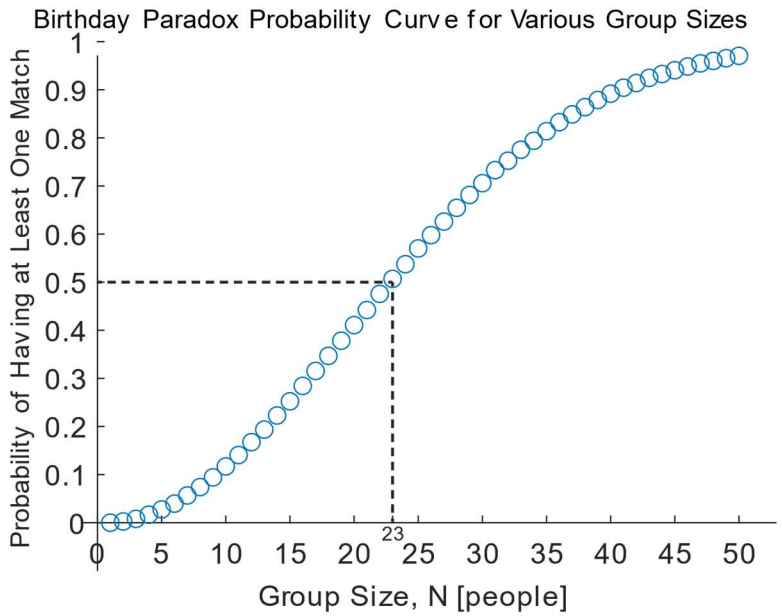

Figure 2 Probability distribution for at least a single matching birthday in a group of particular size.

Disclaimer, this plot uses around 300,000 trials. I only generated this once to have a smooth graph for the report; I found that for plots with only a few thousands of trials the plots still had some deviations. My current code which I submitted runs 3,000 trials for practicality reasons but this can be easily adjusted by changing A at the start of the code. From the plot, we can see that we have around a 50% probability of having at least one pair of matching birthdays in a group of 23 people. We can then compare this to numerical calculations like the following approximation:

$$p(n) = \frac{365!}{365^n(365-n)!}$$

Equation 3.1 Calculates the probability of not having a single pair of matching birthdays in a group of size n.

While most calculators will return an error of too large of a value, since we already have a value in mind, we can simply verify this manually. If we plug in n = 23, then the top factorial will cancel out with the bottom factorial, leaving only $\frac{365*364*\ldots*343}{365^{23}} = 0.4927$. This is the probability of no one in the group matching, so the probability of at least one matching birthday is 1 – 0.4927 = 0.507. This verifies our simulation in Matlab!

3.4 Discussion

The birthday paradox is quite interesting as a case study. One of the major errors I was running into was the fact that we were only looking for a single match. At first, I was counting all of the matches we got over all of our trials, so my probabilities were much higher since some instances would have 3 matches at once which led to upward skewed results. Once I put an upper limit on the number of matches in any one group's simulation, the problem worked itself out and the probability we calculated using Poissons formula was more accurate.