Poker Simulation Update 2020

Karl Kintner-Meyer

December 31, 2020

1 Introduction

I originally set out to write a poker simulation script in the summer of 2017 back when I regularly played poker with friends at the University of Washington. My original goal was to be able to rank starting hole cards in Texas Hold'em Poker as a function of the number of players in the game. In the past, when I had a need to calculate the probability of winning a poker hand, I would use a poker odds website (https://www.cardplayer.com/poker-tools/odds-calculator/texas-holdem). Normally, this worked fine for measuring win percentages of a pre-specified player hand against another pre-specified opponent hand(s). However, there was no functionality that measured the win percentage of a pre-specified hand against an unspecified opponent hand. I eventually found a website (https://homes.luddy.indiana.edu/kapadia/nofoldem/) where a professor in computer science at Indiana University had solved this problem, but I decided that I would be more interested in giving it a try on my own. I decided to create a script that could handle any number of pre-specified cards and would determine the win percentages of different hands via a monte-carlo simulation.

I use several key words commonly throughout this write-up that I wish to clarify here. These key words are: "Hand", "Round", and "Simulation". "Hand" generally refers to an individual player's cards. I will denote the two cards held only by the player as hole cards. The five cards that are communal I will refer to as community cards or board cards. I will refer to a combination of a player's hole cards and board cards as a player's hand. The important point here is that "Hand" is generally player specific. Next, I will refer to a single poker game instance as a "round". In each round, every player has a hand, and we are trying to determine the winner of that round. Finally, I will sometimes refer to the collection of several rounds as the "simulation". As this script is a monte-carlo simulation, I have chosen to simulate one million rounds of poker to ensure an adequate sample size.

Note: I use a lot of poker terminology in this write-up and am assuming that the reader has a good understanding of the rules of poker, the different poker hand strengths, and the tiebreaking rules (which can be found here: https://www.adda52.com/poker/poker-rules/cash-game-rules/tie-breaker-rules). This is also the part of the write-up where I say: "Use this script at your own risk."

2 Original Poker Script (2017)

The original poker simulation script that I wrote in 2017 was a complete mess, partially due to my ill-conceived attempt to write the code on an iPhone while flying to Europe for vacation. Upon reflection, this code had two major issues that stood out to me: lookup table card association and using a bottom-up scoring approach.

2.1 Lookup Table Card Association

To explain the issue with lookup table card association, I need to first give some background on how the script works. When simulating a single round of play, the script generates a random permutation of the integers 1-52 (akin to giving the deck a random shuffle) and then pulls out the number of randomly generated integers that are needed. Recall that the script allows for any number of pre-specified hole cards and board cards, so the script only needs r randomly generated integers, where $r = (2 \cdot NumPlayers + 5) - NumPrespecifiedCards$. I tried self-writing an

algorithm to ensure that none of the integers taken from the random permutation matched a pre-specified integer, but this algorithm was computationally slow, and I ended up using the MATLAB "ismember" command instead. Once the script has a vector with size $(2 \cdot NumPlayers + 5)$ which contains unique randomly-generated integers 1-52 and includes the pre-specified integers, there needs to be a way to associate each integer with a specific card. To do this, I used a lookup table where each integer is matched to a two-character string. The first character in the string is used to denote the card rank and the second character in the string is used to denote the card suit. This follows the common practice in standard poker software packages such as PokerStars to represent cards as two-character strings in text-based hand history files. The card rank characters are numeric for cards of rank 2-9, and capital alphabetic for cards ranked Jack to Ace. Tens were represented with the alphabetic character "T". Suits were represented by lowercase alphabetic characters, "s" for spades, "h" for hearts, and so on. This allows every card to be uniquely identified. For example, a Ten of Spades would be "Ts", a King of Clubs would be "Kc", and a 7 of diamonds would be "7d". The main problem with this approach is that it incurs a lot of "overhead" operations to exchange all the integers for two-character strings, only to have to parse the strings for the rank value and convert it back into an integer-type. Even more problematic is trying to check for cards of the same suit to score flushes and straight flushes. With this method, each string must be broken down into its constituent characters and all of the second characters have to be parsed via if-statements to determine if a flush existed or not. This ends up being a time-consuming process and does not take advantage of MATLAB's matrix-computation architecture.

2.2 Bottom-Up Scoring

The other major issue with my 2017 poker script lay in the methodology I used to score a given hand. The function that I wrote to score hands implements a bottom-up search. It starts by checking a hand for a pair, then a two-pair, then a three-of-a-kind and so on. This means that every single hand that is passed to the scoring function gets checked to see if it meets the criteria for all ten of the possible poker hand strengths. Since poker hands are scored based off the highest hand that exists in the seven cards, bottom-up scoring just does not make much sense here.

3 Poker Script Updates (2020)

In my winter 2020 update for the poker simulation script, I looked to try to eliminate the large computational inefficiencies that the original 2017 script had. The three largest changes I made were: card representation via integer-divide and mod operations, top-down scoring approach, and short-circuit scoring logic.

3.1 Card Representation: Integer-Division and Mod Operations

The first large-scale change I made to the poker simulation script was how the cards were represented. Instead of using a lookup table to associate each integer from 1-52 with a card, I used integer-divide and mod operations to create a 2×1 column vector from each integer. The idea behind using integer-division is that integer-division ignores remainders. This is useful since integer-dividing by 4 splits the integers between 1-52 into groups of four integers which all have the same integer-divide-by-4 value (technically this is not completely true, since the integers 1,2, and 3 will be grouped together but are missing a fourth member, and the integer 52 will be grouped

on its own, but this can be easily solved and will be detailed shortly). As an example, consider the integers 8-11. The integers 8-11 when integer-divided by 4 are all equal to 2, the integers 12-15 when integer-divided by 4 are all equal to 3, and so forth. Within each group of four integers, we note that each integer has a different remainder when divided by 4: either 0,1,2, or 3. We can use the mod-4 operator to extract out this remainder. As a result, each integer can be uniquely characterized by its int-divide-by-4 value and its mod-4 value. This is exactly akin to rank and suit of cards where each card can be uniquely identified by its rank and suit. Thus, for each integer, we say its int-divide-by-4 value is its rank and its mod-4 value is its suit. At this time, I point out the usefulness of having an offset value for the integers. Ideally, what we would like is that 2 is the lowest card rank we can get from using integer-divide. Since the lowest mod-4 value we can get is 0, we realize that we would really like our integers to be in the range 8-59 instead of 1-52. Thus, we simply add 7 to all of our randomly-generated and pre-defined integers and we have accomplished our goal. Once all the integers are in the 8-59 range, it is trivial to extract the ranks and suits for each integer and store them in a matrix with the first row containing the ranks, and the second row containing the suits. As a bonus, this representation circumvents all of the string-parsing inefficiencies mentioned in section 2.1.

3.2 Top-Down Scoring

As mentioned in section 2.2, the bottom-up scoring approach was pretty inefficient, so I changed the structure of the scoring function to check for hand strength starting with the highest possible hand strength and moving down the list until the criteria for a given hand strength is met. Upon meeting the criteria, the scoring function skips the remaining checks for lower hand strengths and returns to the main simulation.

There are also a few other small changes of note here that make a small difference in runtime speed. During the check for a straight flush, the scoring function first starts by checking for a flush, then using the cards in the flush to check for a straight. In this update, I added a flag for the flush check to ensure that even if there wasn't a straight flush, a flush would still be noted and scored as such if no higher hand was found. I also changed the way straights were scored. In the original script, I sorted the hand array from lowest-rank to highest-rank and then used nested if-statements to check whether there were five consecutive cards in a row, starting from the low-rank end of the sorted array and working my way up. Again, this bottom-up approach was inefficient, so I changed it to start the search at the 5th-to-last highest ranked card and work down. I also got rid of the nested if-statements and used multiple && operators to check for consecutive cards.

3.3 Short-Circuited Scoring

The last big change, and my personal favorite, was to add short-circuiting logic to the scoring function. My idea here is best communicated by an example. Consider that the first scored player's hand is a full-house. If we now attempt to score any other player's hand, we only need to check if his hand is at least as strong as a full-house. As such, the short-circuiting logic in the code takes the form of an integer flag that tracks the current strongest hand in the round. If any hand is being scored and doesn't at least meet the strength of the current strongest hand, then that hand is skipped. If a hand is found to be stronger than the current strongest hand, then the integer flag is reassigned to the new value.

I also added a second short-circuiting algorithm that builds on the first one. My idea here stems from the fact that the script is only trying to determine whether the first player won the round or not. One way to check this is by evaluating how many times the current strongest hand flag is reassigned to a new value. By default, it is always reassigned for the first time when the first player is scored. However, if it is reassigned again to a higher value, then that means that another player has a stronger hand than the first player and we can immediately conclude that the first player didn't win.

3.4 Other Notable Changes

I include this subsection to document other small tricks I used in order to speed up runtime. These are not as large of scale as the previously mentioned changes, but I do believe they still make a noticeable difference on the runtime of the script.

The first smaller-scale change of note is how I implemented the monte-carlo simulation trials. In the 2017 poker script, I simply considered all the tasks I needed to do for one round of poker, and then wrapped all the code for those tasks with a for-loop that simulated the round "n" times. While this does work, I thought that maybe there were some tricks to speed this process up. In the 2020 update, I pre-generated all the random integers for all "n" trial runs first and stored them all in a matrix. The computational savings comes in when you eliminate any randomly chosen integers that match pre-defined integers. With the random integers already generated for all "n" rounds, it is easy to zero out all the entries in the matrix that match any of the pre-defined integers. Instead of having to perform the integer-match-check every time you simulate a new round, you can do the check once for all "n" rounds and then be done with it. The trade-off is that this method requires more memory in order to save all of the randomly generated integers for all "n" rounds.

Another small-scale change that I made is eliminating separate tie-breaking and scoring functions. In the original script, I had one function that would score each hand, and then a separate function that would tiebreak any hands that were tied for the strongest hand. In my update, I eliminated the tiebreaking function and had the scoring function return a 1×6 row vector containing both the hand strength and any relevant tiebreaking information. The vector was organized such that the hand strength was saved in the first column, and any relevant tiebreaking values were contained in the remaining five columns. Any unused columns were set to zero. As an example, one might consider a full-house again. A full-house has a hand strength equal to 7 (hand strengths were assigned with high card having strength of 1 and a straight-flush having a strength of 9. There was no need to include a royal-flush as a separate hand as the rules for tiebreaking straight-flushes ensured that a royal-flush would always be the highest). This means that any full-house will have a 7 in the first column of its score vector. Since full-houses are tiebroken first by the rank of the three-of-a-kind and then by the pair, the scoring function puts the rank of the three-of-a-kind in the second column of the hand's scoring vector and the rank of the highest pair (or second highest three-of-a-kind) in the third column. Once every hand in the round is scored, you can stack all the scored row vectors on top of one another into a matrix and go columnwise to evaluate which hand won, using tiebreaking criteria, as necessary. To make this process even faster, I circumvented the column-by-column logic with another trick. Consider the logic of how scoring works if you go column-by-column. You'd first look to see if there was more than one player with the highest value in the first column. If there was, you'd have to look at the second column for only these players, if there were still more than one winner then you'd likewise have to proceed to the third column with

the players still tied for the highest after the second column. Remember that each entry in the first column must be between 1-9 and each entry in the remaining five columns must be between 1-14. If we pretend that the entries in the remaining five columns were restricted to 1-9 then we could pretty easily figure out which row was the winner by creating a six-digit number out of all the entries in a single row. The sixth column would represent the 1's place, the fifth column would represent the 10's place and so on. This would make it very easy to figure out the winner, as once we have turned all the rows turned into six-digit numbers, we could just find which number is the biggest and that would be the winner. Converting each row to a six-digit number is accomplished by multiplying that row by the vector $[10^5, 10^4, 10^3, 10^2, 10^1, 10^0]^T$. Since the entries in the last five columns can range from 1-14 we instead multiply each row by the vector $[10^{10}, 10^8, 10^6, 10^4, 10^2, 10^0]^T$ to ensure that any given column strictly dominates all the columns to its right.

4 Script Walkthrough

I have added a quick summary of how the poker simulation code is structured for those who are interested. I will provide descriptions of each code block as they are annotated in the code.

4.1 Code Block One

This code block serves as the "input" block. This is where the user can specify the number of players in the round, the pre-defined hole cards, and the pre-defined board cards. The pre-defined cards are specified on a scale from 1-52 by the user, and then the script offsets these numbers by 7. The integers are then converted to "uint8" type and for convenience's sake, the rank and suit of the pre-defined cards are generated in their own matrices. Next, we create a matrix that holds all the randomly generated integers for all "n" simulated rounds. Each round's random permutation of numbers gets saved as a row in this matrix. Finally, we offset all the entries in the matrix by 7.

4.2 Code Block Two

This code block zeroes out any entries in the random-permutation matrix that match any of the pre-defined integers.

4.3 Code Block Three

This code block is where most of the integer processing occurs before the scoring can be completed. This code block goes row by row through the random-permutation matrix and pulls out any of the non-zero entries. Since different rows may have different numbers of non-zero entries, it also picks out only the first "m" non-zero entries where "m" is $(2 \cdot NumPlayers + 5) - NumPredefinedCards$. Besides this main task, it also converts the random-permutation matrix to "uint8" format and extracts the rank and suit for all cards in every row.

4.4 Code Block Four

The final code block is where all the scoring happens. The script pulls out one row from the "uint8" format random-permutation matrix, which represents one round of poker with all the pre-defined and randomly generated cards. It progressively runs through each player's hand and scores it by passing it to the scoring function. At the end of the block, the program checks if the first

player is the winner of the round and keeps a count of how many times this occurs. We can then determine the win percentage of a pre-specified hand by dividing the number of player one wins by the number of rounds we have simulated.

5 Results

After the update, I ran both the 2017 version of the script and the 2020 version of the script to simulate one million rounds of poker with the exact same pre-defined hands. The 2017 version script took 449 seconds to simulate the one million rounds, while the new script took only 51 seconds to simulate the same number of rounds. This is approximately an 8.5x speed savings. Debugging the scoring function to ensure that win percentages are accurate is still a work-in-progress, but as of a first-pass attempt, the win percentages seem to be within a percent or so of the win percentages given by various poker odds websites.