Poker Hand Calculator

December 2021

Summary

A fixture on any televised poker game is the calculated odds of each player's hand winning after all the cards are delt.

In Texas Hold'em, each player begins with 2 cards, then, 5 community cards are dealt between rounds of betting. The players' hand is the best 5-card hand formed between these 7 cards.

The probabilities (56%, 44% here) show the likelihood of each player having the best hand across all possible realizations of the 5 community cards.

This is not a new or unique problem; many websites provide these calculations. This was nevertheless an amusing computational challenge:

- The complexity in the hierarchy of poker hands leads to many simple problems; finding pairs and sequences in small sets of numbers.
- The problem is small enough to solve with any modern hardware, but large enough to require a fast and efficient program.

I outline two approaches. The first is conceptually simple and correct but is computationally slow. The second precomputes the key results of the first approach so that results can be returned at a speed comparable to online calculators.

First Approach – Outline

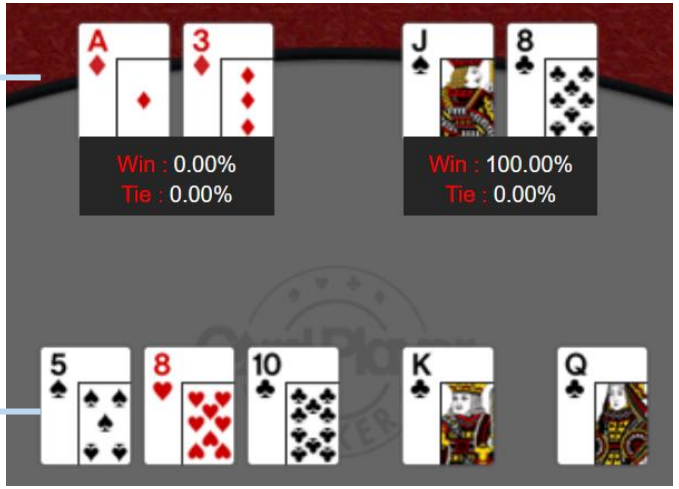
Approach:

- Fix two (or more) starting hands of 2 cards.
- Draw a 5-card set of community cards.
- Determine which of the starting hands forms the strongest hand when combined with the community cards.
- Loop over all possible 5-card sets of community cards and tabulate the frequency at which each hand wins.

The approach requires a function that determines which hand wins for a given set of community cards.

- Input: set of 7 cards (formed from 2 starting cards + 5 community cards).
- Output: numerical score corresponding to the strength of the best possible hand formed from the 7 cards.
- Call the function once for each starting hand.
- Function must be fast; it must be applied to each hand for each of the 1.7 million sets of 5 community cards.

Determine which hand wins for given set of 5 community cards



Tabulate which hand wins over all possible community cards

Scoring Function (hand_rank_main)

Input: set of 7 cards (2 starting cards + 5 community cards).

Output: numerical score corresponding to the strength of the best possible hand formed from the 7 cards.

Function steps:

- Sort cards by value from smallest to largest.
- Calculate the "first differences" for the sequence of card values.
- Use the presence of 0's and 1's to identify straights and pairs:
  - [0]: pair
  - [0,0]: 3 of a kind
  - [0,0,0]: 4 of a kind
  - [1,1,1,1]: straight
  - [1,1,0,1,1]: straight and pair
  - ...
- Assign a score according to hand strength
  - Integer value for hand: 1,2,3...8 for pair, 2-pair, 3-of-a-kind, ..., straight flush.
  - Decimal values for the strength of cards
    - Example: pair of 3s and 4s, with 10 high card: 2.040310
    - Example: full house with aces and 8s: 6.1408

Illustration of first differences calculation. Example hand: first difference reveals that there are no pairs or straights.

```
calculate the "first difference" array by
shifting index by 1 and substracting

ex. hand = [2,3,6,7,9,12,14]
  [2,3,6,7, 9,12,14]
[2,3,6,7,9,12,14]    -
--------------------------
= [1,3,1,2, 3,2]

the appearance of combinations of 0s or 1s
indicates certain hands:

[1,1,1,1] => straight
[0]       => pair
[0,0]     => 3 of a kind
[0,0,0]   => 4 of a kind
[0] in 2 places => 2 pairs
```

Computation

Coding done using the Julia programming language. Benefits of Julia:

- Oriented towards fast numerical calculation through base library and compilation.
- Simple syntax like other high-level languages plus support for Unicode and LaTeX symbols.
- Easy parallel processing: single line of code to run for-loops on all CPU cores.
- Supports custom types (structs, objects), used to define a "card" type.

```
17  struct card
18      number::Int
19      suit::Int
20  end
```


```
# 1. starting hands
hand_1 = [card(14,2), card(3,2)]
hand_2 = [card(11,3), card(8,4)]
```

```
# 1. starting hands
hand_1 = [card(A,♢), card(3,♢)]
hand_2 = [card(J,♠), card(8,♣)]
```



```
# 2. build the deck of remaining cards
for number in 2:14
    for suit in 1:4
        # add card to deck
        deck[card_number] = Card(number, suit)
        card_number += 1
    end
end
```

```
# 3. build all possible 5-card boards
boards = collect(subsets(deck, 5))
```



```
# get score of each hand for given board of 5 cards
h1_score = hand_rank_main([hand_1; boards[b]])
h2_score = hand_rank_main([hand_2; boards[b]])

# record which hand won
if h1_score > h2_score
    h1_wins[b] = 1
end
```

Computation

Coding done using the Julia programming language. Benefits of Julia:

- Oriented towards fast numerical calculation through base library and compilation.
- Simple syntax like other high-level languages plus support for Unicode and LaTeX symbols.
- Easy parallel processing: single line of code to run for-loops on all CPU cores.
- Supports custom types (structs, objects), used to define a "card" type.

```
17  struct card
18      number::Int
19      suit::Int
20  end
```

```
# 3. build all possible 5-card boards
boards = collect(subsets(deck, 5))
```

```
# 1. starting hands
hand_1 = [card(14,2), card(3,2)]
hand_2 = [card(11,3), card(8,4)]
```



```
# 1. starting hands
hand_1 = [card(A,♢), card(3,♢)]
hand_2 = [card(J,♠), card(8,♣)]
```

```
# 2. build the deck of remaining cards
for number in 2:14
    for suit in 1:4
        # add card to deck
        deck[card_number] = Card(number, suit)
        card_number += 1
    end
end
```



```
# alternative silly Julia syntax
for number in [2:10,J,Q,K,A]
    for suit in [♡,♢,♠,♣]
        # add card to deck
        deck[card_number] = Card(number, suit)
        card_number += 1
    end
end
```

Sample Code

```
# B. function to check if vector of cards contains a flush and return corresponding indices
# returns the index of all flush cards, not just the largest
function get_flush_indices(cards)
    # get suit of each card
    suits = [card.suit for card in cards]

    # counts of each suit
    counts = [count(==(suit), suits) for suit in 1:4]

    # check if there is at least 5 of 1 suit
    if maximum(counts) >= 5

        # flush suit
        flush_suit = argmax(counts)

        # return indices of cards in flush
        flush_index = findall(==(flush_suit), suits)

        return flush_index
    else
        # if there is no flush, still reutrn Vector{Int64} but with no indices
        return [0]
    end
end
```

First Approach – Results

Calculated win probabilities are identical to those produced by online calculators.

Calculation time is approximately 15 seconds for 2-player hands and 20 seconds for 3-player hands (using multi-threading on 6-core CPU). The time does not scale linearly because each additional hand reduces the number of possible sets of community cards.

Calculation time for the online calculators is essentially instantaneous. I do not know if the online calculators perform an ad hoc calculation or look up the probabilities from a precomputed table.

Computing the win probabilities for all possible pairs of starting hands would unfortunately take ~3,300 hours using my program and hardware.

Probabilities from cardplayer.com

Probabilities calculated using my program for the same starting hands

```
# 5. get results
h1_win_percent = mean(h1_wins) | 0.5946070323961166
h2_win_percent = mean(h2_wins) | 0.4011454741681384
tie_percent = 1.00 - h1_win_percent - h2_win_percent | 0.004247493435745042
```

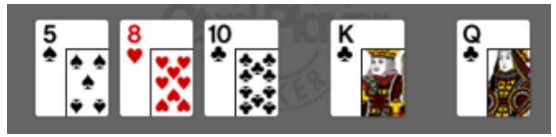
Second Approach – Outline

Unsatisfied with the correct but slow calculation from my first approach, I sought to match the speed of the websites that perform these calculations using a precalculated table of hand scores.

Second approach

- Fix a 2-card starting hand.
- Calculate the hand score for all 2.6 million possible sets of 5 community cards.
- Repeat for all 1,326 starting hands.
- A hand's win percentage is the fraction of boards where it scores higher than competing hands.

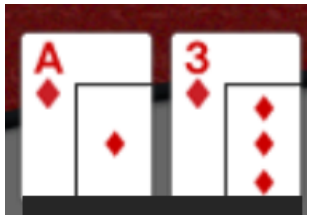
Table showing the hand score for each starting hand under each possible set of 5 community cards (Boards). Table is precomputed before calculating probabilities.



Index of all sets of 5 community cards (boards)

Starting Hand	Boards			
	1	2	...	2,598,960
1	3.021110	2.100513	...	7.0214
2	2.110504	4.08	...	2.050403
...	...	...	...	...
1326	3.051302	Invalid	...	7.0213

Index of all 2-card starting hands

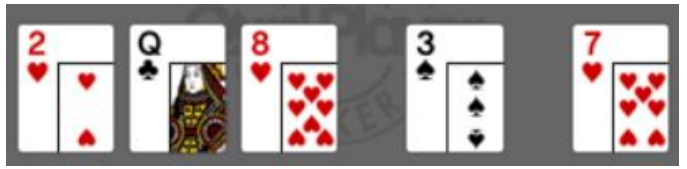


Score for each starting hand given board #1

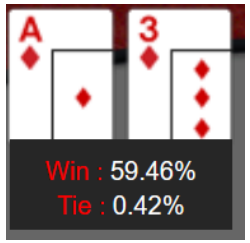
Some combinations of hands and boards are invalid because the same card is used in both the hand and the board. These are excluded from the calculation.

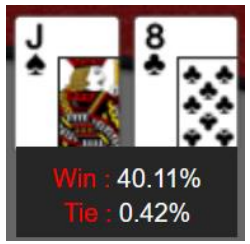
Second Approach – Illustrative Example

Board 100



Board 500,000





Starting Hand	Boards									
	1	...	100	101	...	500,000	500,001	...	1,712,303	1,712,304
304	7.0214	...	6.0203	6.0203	...	1.03141208	1.03141208	...	7.1413	7.1413
1265	7.0214	...	3.021108	3.021108	...	1.08121107	1.08121107	...	6.1413	6.1413

- Hand 304 wins in 59.46% of the boards (0.97 million out of 1.7 million)
- Hand 1265 wins in 40.11% of boards (0.69 million out of 1.7 million)
- The hands tie in 0.42% of boards (0.07 million out of 1.7 million)

- Hand 304 makes a full house: (three 2s and two 3s).
- Hand 1265 makes only 3-of-a-kind (three 2s).

- Hand 304 makes a pair of 3s.
- Hand 1265 makes a pair of 8s.

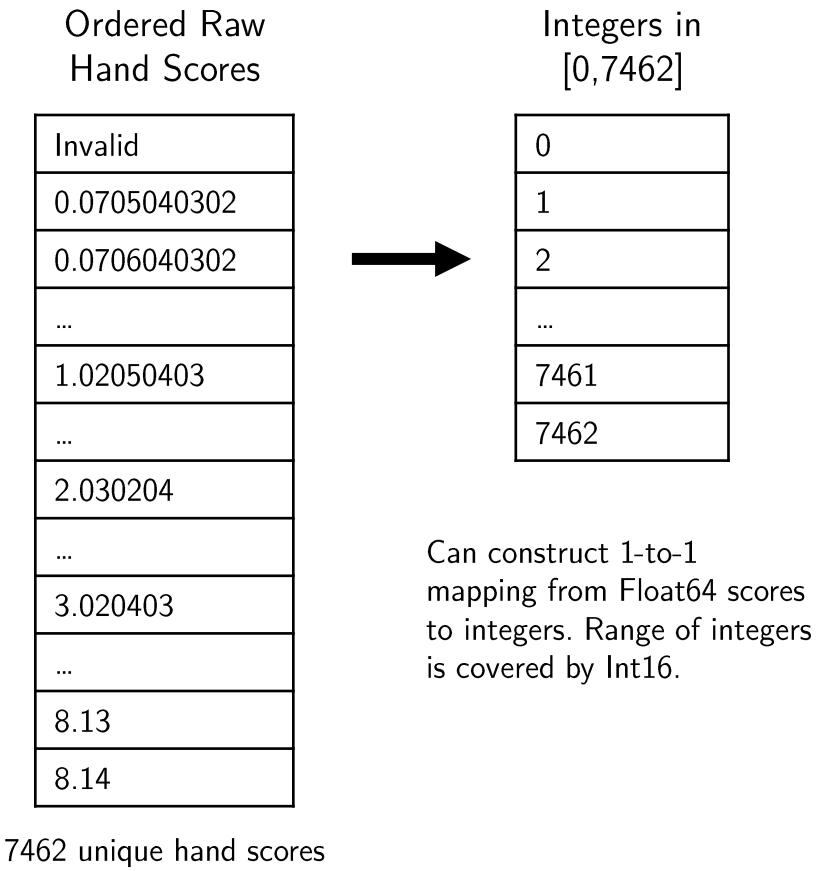
Note: only 1.7 million out of 2.6 million boards are valid (no duplicate cards). These are all the 5-card subsets of the 48 remaining cards of the 52-card deck.

Second Approach – Data Issue

Storing the raw scores in a single table creates a 30 GB array. I explored two methods of reducing the size of this table to approximately 5-7 GB such that is can be loaded into memory on typical PC.

- There are only 7462 unique possible hand scores, so Float16 or Int16 contain enough unique values to store hand scores if they are mapped into a suitable range. The size of the table can therefore be reduced by a factor of 4.
- There is some level of redundancy between hands and suits. For example, the hand $(3\spadesuit, 4\spadesuit)$ spades has the same win probability against $(5\diamondsuit, 6\diamondsuit)$ and $(5\heartsuit, 6\heartsuit)$.

Alternatively, the table can be split into rows corresponding to unique starting hands. When the hands are selected, the relevant row of 2.6 million scores for each hand is loaded into memory. Doing so in Julia was prohibitively slow (4.3 seconds to load CSV files), but was suitably fast in Python (0.5-1.0 seconds to load CSV files, or ~0.3 seconds to load from pickle files).



Second Approach – Results and Comparison

This second approach rivals the near instantaneous speed of the web-based calculators when the relevant data is already in memory or can be loaded suitably fast.

- Tabulating the win, loss and tie percentage for each pair of hands takes only ~5 ms.
- The total time (loading the data and tabulating probabilities) takes 0.3 seconds with Python's pickle package, 0.5-1.0 seconds with Python's pandas package, and 4.3 second with Julia's CSV package.
- The speed of the web-based calculators is approximately 0.1-0.3 seconds (estimated by inspection).

Looking at the code on the webpage for one of the web-based calculators, the interface communicates with a server that is responsible for performing the calculation.

Therefore, this second approach could be identical to that employed by these websites and, at minimum, offers comparable speed.

Method		Total Calculation Time
First Approach	Julia	13-15 seconds
Second Approach	Julia (CSV)	4.3 seconds
	Python (pandas)	0.5-1.0 seconds
	Python (pickle)	0.3 seconds
Web-based calculator	Web interface + server?	0.1-0.3 seconds

```
(i.prototype.calcOdds = function () {
        var t;
        return (
          (t = this.jsonGame()),
          t.seats.length >= this.settings.min_hands
            ? $.ajax({
                url: "/poker-tools/odds-calculator/api",
                dataType: "json",
                data: t,
                success: this.parseResponse,
                error: this.handleError,
              })
            : void 0
        );
```