Decima: a Poker Simulation Framework

By Chris Kottmyer and Shahin Shirazi

Abstracts:

Decima is a Texas Hold'em Poker Simulation Framework written in Python allowing users to rapidly prototype and design poker strategies, which are then used in simulations of 2-6 player Poker games. The simulator is highly extensible and configurable, allowing users to set up multiple simulations, change the number of replications (poker tables) and number of hands played at each table. Poker player strategies are designed by inheriting from a Python class called GenericPlayer and extending it with a single method: bet_strategy to implement player betting strategies. We utilized Decima to evaluate two poker strategies by comparing them in 2,3,4,5 and 6 player games against opponents that always called or checked their bets. We compared the player's performance by looking at their average gain/loss per game, their mean balance over 100 hands averaged across 30 replications, their predicted probability of winning based on Monte Carlo Simulation vs actual poker final hands and their strategies concerning pre-flop win, loss and fold rates. We found one player strategy, dubbed Smart Player, consistently won poker games and tended to make a consistent profit over time. Another player, Conservative Player, was more selective about cards he/she played, but made larger bets leading to higher variability in wins than smart player. Both players gained chips over time, but Conservative Player gained chips at a faster rate. Finally we ran quick simulations with all 6 different strategies to identify best performing strategy. To our surprise our Conservative Player outperformed Smart Players by order of magnitude. A player strategy that bet in portion to his odds of winning won almost the same amount as Smart Player, which indicates the importance of betting strategy on overall gains.

Background and Description of problem:

"The development of probability theory in late 1400s was attributed to gambling; when playing a game with high stakes, players wanted to know what the chance of winning would be"[1] . Like most gambling games, probability plays an important role in 7 card Poker. There are 133,784,560 ways to draw a given hand out of 52 cards in standard deck. Using simple probability formula, we can calculate the probability of getting specific hand. For example there are 4,327 different ways to draw a royal flush, therefore one would expect to draw this hand once every 30,940 draws, or 0.0032% of the time.[1] Although calculating probability of each hand is a simple and important factor in poker games, it is not enough. Probability of getting a specific hand for any specific player constantly changes depending on their current hand, number of players involved, how many cards have been dealt, etc. Using math to calculate the probability of every possible hand at any given moment during the game can be cumbersome. Simulation can help us to evaluate different scenarios without going through detailed probability theory to get the answer. The caveat to this solution is that if we don't run enough trials, we may not get the correct answer. To avoid this risk we checked the probability of different hands from our simulation against actual probability for the given hand using probability theory.

This simulation assigned each player to a specific table and kept track of each player at any given table for any specific hand to assist in statistical analysis. During Conservative and Smart players betting decisions, a Monte-Carlo simulation is run for 100 iterations to estimate winning odds for each hand. It takes the player's current hand, number of opponents and current number of cards left in the deck to check the winning odds for current player. Conservative and Smart player are two poker strategies we developed when building Decima.

Definitions:

Win probability: Calculated probability of winning a hand based on current hand and community cards. This probability is calculated by running 100 iterations of Monte Carlo simulation of the players current hand, community cards and number of opponents playing. It can be summarized as the percent of wins a player received after playing the current hand and community cards 100 times (ties are counted as wins).

Equal chance probability: Probability of choosing a player at random to win the game. This is equivalent to rolling a N-sided dice to assign winning position. Where N is the number of players.

Community cards: cards that placed on table and everyone can incorporate in his/her hand. Community cards are not used during pre-flop and during post flop the dealer begins by showing 3 community cards. He/She then flips one additional card for the next two phases for a total of 4-community and 5-community cards respectively.

Poker phase: We define a poker phase as either: pre-flop, 3-community card, 4-community card or 5-community card portion of a poker game. A phase is composed of 3 betting rounds where each player can make a bet assuming they did not fold in previous betting rounds.

Betting round: We define a betting round as a round where all non-folded players have an opportunity to fold, check, call or raise the bet. The 3rd betting round in our version of poker does not allow for raises.

Active Player: A poker player that has not folded during a poker game. Active players continue to make bets until they fold and become inactive. If an active player reaches the end of the game, his hand is scored and a winner is determined. A poker game winner is always an active player.

Player Strategy: A set of rules that defines how a player makes a bet. Player strategies can be shared by multiple players.

Player: An agent at a poker table that uses a player strategy to make decisions. Multiple players can exist at a table using the same or varying strategies. Each player is independent of other players retaining his own balance and receiving his own set of cards.

Scenario: A set of simulations that have a common theme. A good example would be testing a single strategy by running a set of 5 simulations: 2,3,4,5 and 6 player tables to see how the

agent performs when faced with different number of opponents. In our analysis, we run two scenarios to test two different player strategies: Conservative and Smart player.

Pot: The sum total of all player bets that a player can win at the end of a poker game.

Problem statement:

Gaining poker experience and testing out poker strategies can take hundreds of hours over multiple days, weeks or even years. Since poker is a form of gambling, gaining experience can cost significant amounts of money. Our main goal was to develop a software application that aids users in quickly prototyping poker strategies, determine what strategies led to the highest profit and through detailed simulation analyze the strengths and weaknesses of those strategies.

Installing and Running Decima:

Running Decima: All you need to run Decima is Python 3. Python 3.7 was used for this simulation; however other version of Python 3 should be compatible with our code. Python 3 is typically installed on Linux and MacOSX systems. For Windows users, Python 3 can be download from https://www.python.org/downloads/. All the libraries that were utilized in Decima come pre-packaged with Python 3 to simplify installation.

The source code for Decima can be downloaded from our GitHub account: https://github.com/Silber8806/PokerPro. Once downloaded, go to the directory: source_code and type python poker.py to run our simulation. This can take upwards of 2 hours. poker.py runs a total of 21 simulations with 30 replications (poker tables each) and 100 poker games per table. It compares two strategies: Conservative Player and Smart Player by setting up 2,3,4,5 and 6 player poker tables and pitting one of the aforementioned agents against players that always call or check bets. One additional simulation that used all player strategies we developed. After Decima is finished running, a folder: source_code/data is created with 3 types of files. These files are: poker_balances_[table_id].csv, poker_hands_[table_id].csv and poker_info_[table_id].csv where [table_id] represents a unique integer identifying a poker table. These files store information about player balances, player bets and high-level table information.

Note: re-running the simulation does not delete information in source_code/data folder. We suggest deleting the source_code/data folder after each run.

Configuring Decima and changing simulation parameters:

Poker.py is the only file required to run Decima. It contains two sections that user can manipulate to change the simulators behavior. First is the simulation variable, which tells Decima what simulations to run (Fig.1), how many poker tables to run for each simulation (tables), how many poker games to play for each table (hands), the starting balance of each player (balance) and the minimum balance to join a game (minimum balance). Top four parameters (tables, hands, balance, minimum_balance) apply to all simulation scenarios. For example, if you set tables to 100 and hands to 50, it will run 100 poker tables and 50 games per poker table for a grand total of 5,000 poker games per simulation. The simulations

key is a list of simulations to run. A simulation in this list, has two keys: simulation_name which is a human friendly name for the simulation and player_types which includes list of player strategies to use in the simulation.

The default configuration used for our analysis is a set of 11 simulations split into two groupings: Smart Player and Conservative player simulations. Each group has 5 separate simulations representing: 2, 3, 4, 5 and 6 player games putting a single Smart/Conservative Player against a group of Always Call Players. We also ran 1 additional simulation utilizing 6 different player strategies. For our simulation, we set tables to 30, hands to 100, balance to $100,000 and minimum balance to join a table of $50.00. This represents 11 simulations, 330 poker tables and 33,000 games of poker. Each poker table's players represent individuals playing poker and are independent of each other outside of using similar poker strategies when playing poker.

```
1357  debug=0 # to see detailed messages of simulation, put this to 1, think verbose mode
1358
1359  if __name__ == '__main__':
1360      print("starting poker simulation...(set debug=1 to see messages)")
1361
1362      # defines all the simulations we will run
1363      simulations = {
1364          'tables': 10, # number of poker tables simulated
1365          'hands': 100, # number of hands the dealer will player, has to be greater than 2
1366          'balance': 100000, # beginning balance in dollars, recommend > 10,000 unless you want player to run out of money
1367          'minimum_balance': 50, # minimum balance to join a table
1368          'simulations': [ # each dict in the list is a simulation to run
1369              {
1370                  'simulation_name': 'smart vs 1 all call player', # name of simulation - reference for data analytics
1371                  'player_types': [  # type of players, see the subclasses of GenericPlayer
1372                      SmartPlayer, # defines strategy of player 1
1373                      AlwaysCallPlayer, # defines strategy of player 2
1374                      AlwaysCallPlayer, # defines strategy of player 3
1375                      AlwaysCallPlayer, # defines strategy of player 4
1376                      AlwaysCallPlayer, # defines strategy of player 5
1377                      AlwaysCallPlayer # defines strategy of player 6
1378                  ]
1379              },
1380          ]
1381      }
1382
1383      random.seed(42) # gurantees standardized output for any given config
1384
1385      run_all_simulations(simulations) # runs all the simulations in simulation variable
1386
```

Fig.1: Selecting simulation variable

Defining Player Strategies to use in Decima:

The second way user can manipulate Decima is by defining poker strategies. To create a new player strategy, first define a Python class that inherits from GenericPlayer and implements the bet_strategy method and then add the class (not instance) to a simulation's player_types list. See Fig 2.

```
# sophisticated player with more complicated strategy.
class SmartPlayer(GenericPlayer):
    def bet_strategy(self,hand,river,opponents,call_bid,current_bid,pot,raise_allowed=False):
        win_probabilty = simulate_win_odds(cards=hand,river=river,opponents=opponents,runtimes=100)
        expected_profit = round(win_probabilty * pot - (1 - win_probabilty) * current_bid,2)
        equal_chance_probability = 1 / float(opponents + 1)
        high_probability_of_win = 1.4 * equal_chance_probability

        if win_probabilty > high_probability_of_win:
            self.raise_bet(100)
            return None

        # if you statistically will make money, call the bid else just fold
        if expected_profit > 0:
            self.call_bet()
        else:
            self.fold_bet()
        return None

# Player that always calls
class AlwaysCallPlayer(GenericPlayer):
    def bet_strategy(self,hand,river,opponents,call_bid,current_bid,pot,raise_allowed=False):
        self.call_bet()
        return None
```

Fig.2: Defining a player strategy.

The GenericPlayer class provides common functionality for all player strategies. This includes interacting with poker games (make_bet method), default action to call/check when a player is the only person not to fold representing an automatic win for the player and 3 methods used to signal a betting decision: fold_bet(), call_bet() and raise_bet(). It also has mechanisms for recording a player's decisions for simulation analysis. A Decima user should avoid modifying this class, but should be aware of what functionality it provides.

Once a user has created a class that inherits from GenericPlayer, it needs to create a bet_strategy method. Bet strategy represents a players decision making process when deciding to fold, check, call or increase the bet (raise). It is typically written in Python in the body of the bet_strategy method. The bet_strategy method needs to implement all the following parameters, which the user is encouraged to use within the bet_strategy method:

- **hand:** represents the two-card hand a player receives during pre-flop round. The hands are Python named tuples: Card(rank, suite) in the form of a list. You can access suite and rank using dot notation: card.rank and card.suite.
- **river:** represents 0,3,4 or 5 community cards. It uses the same Python named tuple: Card(rank,suite) in list format. Cards that have yet to be dealt are represented by None.
- **opponents:** an integer representing the number of opponents still playing the game.
- **call_bid:** an integer representing the amount of money needed to match the current bid.
- **current_bid:** an integer representing the amount of money the player has already bet.
- **pot:** an integer representing the money contributed by all players to the pot or reward.

- **raise_allowed:** a Boolean value indicating if raises or increasing a bet is allowed. When this is set to False raise_bet method will use call_bet under the cover.

To make a bet decision, a user has to call one of these functions once and only once: fold_bet(), call_bet() or raise_bet(raise_amount). The functionality is explained below:

- **fold_bet():** the player folds his hand and doesn't play for the remainder of the game.
- **call_bet(allow_all_in=True):** a player matches the current bid (increases players bet by call_bid) and continues playing the game. allow_all_in deals with the situation where the call_bid amount is greater than his balance. If set to True, the player will use all chips possible in the bet, if set to False the player will fold instead. The default is True.
- **raise_bet(raise_amount, allow_all_in=True):** a player matches the current bid and then increases it by [raise amount] chips. The allow_all_in parameter determines if a player will use every remaining chip to fund this bid or fold instead. This mirrors the call_bet methods allow_all_in parameter. The default is True.

Note, we didn't get around to writing a guarantee that one of the above is called once and only once. This is a feature that will be implemented soon.

How Decima is implemented:

Decima uses the process-interaction simulation paradigm. We implement the following entities in Decima:

- **Card:** a Python named tuple Card(rank, suite) representing one of the 52-cards in a poker deck.
- **FrenchDeck:** a 52-card deck that represents a dealer. The dealer deals a hand to a poker game, completely reshuffles the deck and then serves the next hand to the next poker game.
- **Table:** a poker table representing a game between a set of players. Sets up a dealer (FrenchDeck) and starts the games (Game entity). A poker table is disposed when all poker games are played. Once all games are completed, the table is responsible for writing out all simulation outputs.
- **Game:** encompasses a poker game and all its rules. A poker game enforces a minimum balance to join and automatically stops if only 1 player has the required balance to join. Each poker game consists of 4 phases: set-up, pre-flop, post-flop and scoring steps.
- **Players:** represents a player. A player interacts primarily with the Games entity by making betting decisions (make_bet, which calls bet_strategy). Players are created by Table entities, which configure the poker games (Game entities).

When Decima begins it validates the simulation variable for errors. If none are found, it sequentially goes through all simulations generating in parallel Table instances. Table instances act independently (parallel processing). They create players based on the simulation configurations player_types list, start a FrenchDeck and then feed hands from the FrenchDeck to Game instances.

After each game, it shifts the player order by one. Once all games are finished, it writes all simulation outputs to the source_code/data directory using its unique table_id to differentiate itself from other tables running concurrently. Decima finishes when all simulations have run all their tables. The process can be seen in Fig.3.

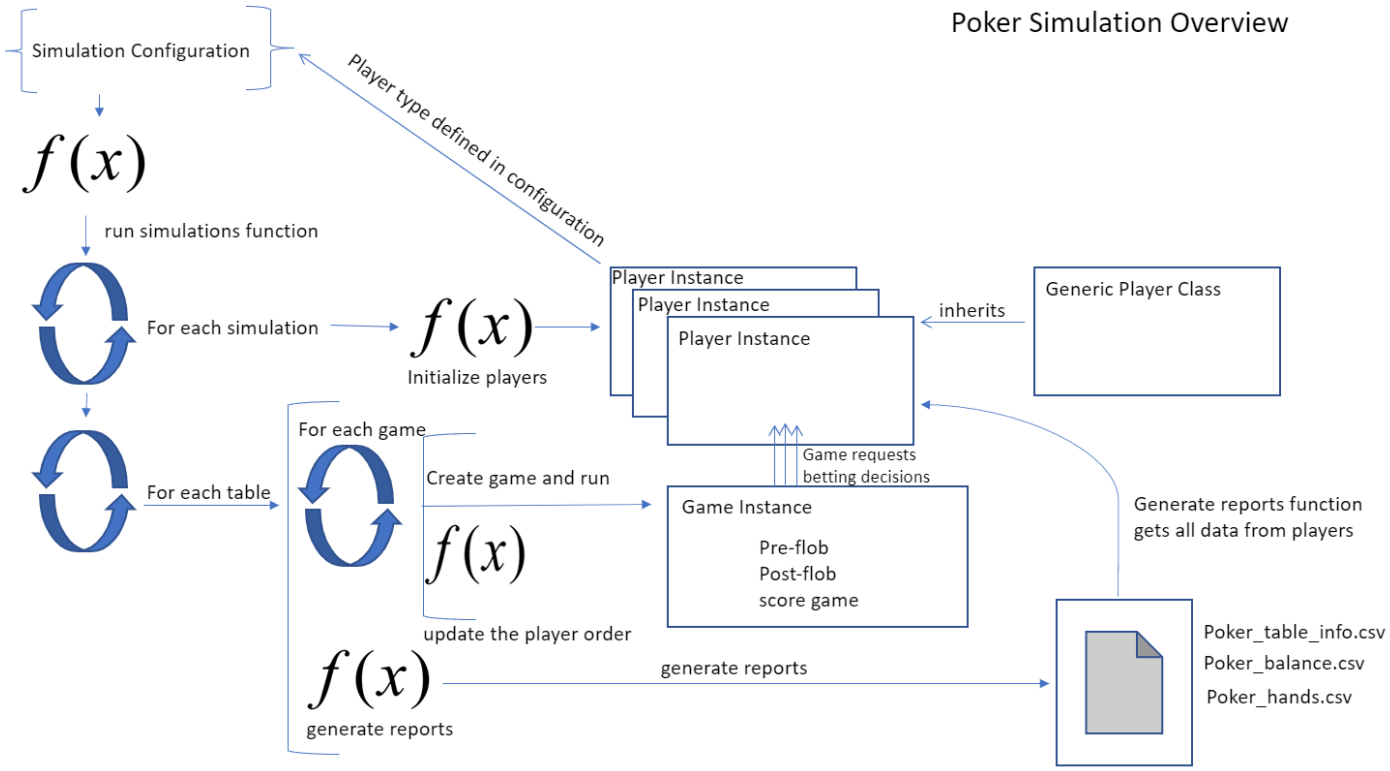

Fig.3: Decima structure.

It's worth mentioning some details about the Game class. The Game class represents a poker game. It has four major steps: set-up, pre-flop, post-flop and game scoring. The set-up step filters out players that don't have the minimum balance to play and ends the game pre-maturely if only 1 player has the minimum balance. It also sets up a small and big blind ($10 and $15 for the last two players respectively). It also creates data structures to record information that is only used in that specific game (like players final hand: high card, pairs). After the set-up, the game starts the pre-flop phase. The pre-flop phase starts by dealing 2-cards to each player and allowing the players in-turn to bet up to 3 times on those cards (no raises allowed on 3rd round). If all players call, the betting rounds end pre-maturely. If all players attempt to fold, the game ends maturely with the player unable to fold winning. After the pre-flop round, the turn order is arranged such that the small and big blind players get to go first (per poker rules). The post-flop phase then begins. The players go through 3 different phases: 3,4 and 5-community card phases each with 3 betting rounds. The post-flop round can pre-maturely end if all, but one player folds. A betting round can end pre-maturely if all players call. This is similar to the pre-flop round. The last stage is scoring phase. In this phase, the Game instance scores each active player's hand (those that have not folded), determines winners and then gives the winners their reward.

Players interact with the Game in only three ways. They can register for games, they can make bets and receive an award. Game registration is used to reset variables mostly important in simulation outputs

stage. Players make bets when a Game calls their make_bet method, which in-turn calls their bet_strategy method. make_bet returns either None representing the player folding or a number representing the updated total bet value. This is handled by the: fold_bet, call_bet and raise_bet methods. The Game then uses the new bet information to update its own state. The Game instance during the scoring stage calls a players get_pot method to give a player the pot.

Monte Carlo simulations:

To estimate winning odds of each player Monte Carlo simulation was implemented (simulate_to_win_odds function in the code) to run 100 iteration of different hands and calculate the winning probability. This function takes player cards, community card and number of opponents as input, creates a deck of 52 cards, removes player and community cards from the deck and simulate the poker game for 100 time to come up with winning odds for the player.

Verifying Decima is working:

Decima was verified against poker odds found on wikipedia mentioned in previous sections. Probability calculated using Decima simulation came very close to the one calculated from probability theory, therefore we are confident that our simulation ran through enough iterations. (table 1).

Decima also has a debug mode (poker.py parameters set to debug=1 and use_parallel=0) that allows a user to see some of the interactions between the Game entity and players. This was used to make sure poker games were following the appropriate rules and player balances were being updated properly. Debug mode requires use_parallel parameter to be set to 0 since forcing Decima into a much slower serial mode. When use_parallel is set to 1 many Games are played concurrently causing many games to post messages at the same time (hence why it's disabled).

Poker Hand	Decima Calculated Probability	7 Card Poker Probability[1]
flush	3.38%	3.03%
four_of_kind	0.19%	0.17%
full_house	2.97%	2.60%
high_card	15.79%	17.40%
one_pair	42.46%	43.80%
straight	4.93%	4.62%
straight_flush	0.04%	0.03%
three_of_kind	5.29%	4.83%
two_pair	24.95%	23.50%

Table.1. 7 card Poker hand probability. Decima vs

Decima outputs:

The simulation generates 3 files for each table: poker_balances_[table_id].csv, poker_hands_[table_id].csv and poker_table_info_[table_id].csv. poker_balances csv rows represents how a player was impacted by a specific game. It includes information about the result of the game: win, loss, folded or last man standing, if the player was small/big blind, the result of his hand: high card, 2-pair, straight and any changes to the players balance as a result of the game. poker_table_info csv provides information about how a table is associated with a scenario and includes things such as the scenario name and player classes used. poker_hands csv lists every single betting decision made by a player and allow information that player had available at the time of the bet. This includes things like: number of opponents actively betting, the current pot/reward, his current bet, how much to call the bet, if raises where allowed, the poker players final bet, their current hand and what community cards are visible. It is worth noting when a player folds he no longer generates betting information. A player however can bet up to 3 times during a pre-flop or post-flop round. This is visible within the dataset. Data can be analyzed over all simulations by aggregating each csv type across all tables. Common keys: table_id, game_id, player_name and player_type are provided as a way to create SQL like joins between poker_balances, poker_hands and poker_table_info csv files, which is used in our analysis.

Our Decima Scenarios and Analysis:

Decima ran through 3 scenarios, 11 simulations and 6 different player strategies to identify if there is a universal strategy that outperforms others in all scenarios or if a specific strategy has gained prominence in certain situations. In the following section, you will find details on each strategy and scenario. We selected a high initial balance of $100,000 compared to significantly lower bet amounts to make sure we could reach a steady state.

Strategies.

Always Call Strategy- A simple strategy that always calls the bet and plays every single hand.

Always Raise Strategy- This strategy always raises the bet by $10 and plays every single hand.

Calculated Player Strategy- Strategy that uses Win Probability (100 Monte Carlo simulations) to estimate the winning odds based on his current hand and compares this against the Equal Chance Probability. If Win Probability is greater than or equal to Equal Chance Probability, player calls the bet otherwise he folds.

Gamble by Probability Strategy- Follows the same strategy as Calculated Strategy, the only difference is that in this strategy player he raises the bet proportional to his win percent instead of just calling the bet. (i.e: if the win percent is 60% player rise the bet by $60). As with Calculated Strategy, the player will fold in when his win probability is less than the "Equal Chance Probability"

Smart Player Strategy: In this strategy player raise the bet by $100 if Win Probability is higher than Equal Chance probability by 40%, otherwise Smart Player calculates expected profit (expected profit=Win Probability *Pot -[1-Win Probability]*current bet). If expected profit is positive, it calls the bet otherwise it folds it's hand.

Conservative Player Strategy- Unlike previous strategies player doesn't compare its Win Probability against Equal Chance Probability, instead it uses following rules based on its Win Probability. Fold if its less than 50%, call the bet if it's between 50% and 70%, raise the bet by 1% of his current balance if Win Probability is between 70% and 90%, raise the bet by 2% if it's between 90% and 95%, raise the bet by 5% if it's between 95% and 99% and finally raise the bet by 10% if its Win Probability is above 99%.

Caveat of Conservative Player Strategy:

Conservative player is very susceptible to its betting limit. Finding the optimal betting amount and decision limit by itself can be the topic for another optimization/simulation problem. We tried to simplify this process by running Monte Carlo simulations against 2 players in order to be able to set constant limit for different betting strategies. We started by setting the folding limit at 70% , which resulted in Conservative player folding every single hands. Then we set the folding limit at 50% and used a more aggressive betting strategy (i.e: all in if percentage was above 99%), which introduced huge variations and inconsistencies. Conservative player was winning the table in some games and losing in others. Using some trials and errors we came up with a final limit, which made the result more consistent while keeping the betting on the aggressive side.

Scenarios:

Always Call Strategy vs Smart Player Strategy: In this scenario we stacked Always Call player against Smart Player and ran different simulations by changing the number of Always Call Player from 1 to 5 (5 scenarios) to check if number of Always Call Player makes any difference on the outcome of Smart Player balance.

Always Call Strategy vs Conservative Player Strategy: In this scenario we stacked Always Call player against Conservative Player and ran different simulations by changing the number of Always Call Player from 1 to 5 (5 scenarios) to check if number of Always Call Player makes any difference on the outcome of Conservative Player balance.

Smart Player Strategy vs all other Strategies: Utilized all the strategies that was created (Always Rise, Always Call, Calculated Player, Gamble by Probability, Smart Player and Conservative Player) to check the outcome and find out the best strategy amongst them all. Due to time limitations we didn't run extensive analysis for this scenario compared to the first two scenarios, however this simulation did help us to identify the best strategy.

Note: There are thousands if not millions of combinations of different strategies and scenarios that can be studied, we limited our research to the above scenarios due to time constraints.

Running the Analysis outputs:

We aggregated the outputs in Jupyter notebooks called: "Poker Simulation Analysis.ipynb", which can be found in source_code/analysis folder. We have left this file unaltered with all cells executed for viewers to easily see our charts and analysis. If you would like to conduct further analysis unzip the

data.zip file into the source_code/analysis/data folder and use our Jupyter notebook to explore the data set. Dependencies include: Python 3.3, Pandas, Numpy and Seaborn python libraries.

Comparison of Conservative and Smart Players:

During our analysis, we generally looked at the outcomes of Conservative and Smart player compared to their opponents the always call player. We focused initially on analyzing the mean number of chips won (mean number of chips is the average number of chips won at a given table, we analyzed 30 means per simulation) or lost during a typical game, the balance over 100 games for the average player strategy and the odds of them winning based on their initial set of two cards. We show how the simulation outputs can be used to investigate and study poker strategies developed with Decima.

Mean Game Balance Net Change:

To determine whether Conservative Player or Smart Player did better, we decided to look at the mean number of chips gained or lost per game. A high mean would generally indicate a better strategy. Since we ran 10 simulations for the first two scenarios, we can plot each of the 30 table means in their own histogram(Fig.4). Each row in Fig.4. represents a simulation with the always call players histogram being in the left column and the Smart or Conservative player being presented on the right column. The top 5 rows represent the conservative player scenarios, while the bottom 5 represent the Smart player scenarios. Within each scenario, the histograms are ordered in ascending number of players with total number of players indicated after type of player (i.e: SmartPlayer-6 shows the balance for Smart Player when he played against 5 Always Call player). This allows us to see how players did in their specific simulation, spot trends associated with increased number of players and compare the two scenarios with each other.

Looking at Fig.4., we notice that Conservative player scenarios (top 5 rows) exhibit extreme variability in betting gains/losses. At many of the tables, Conservative player loses on average some money per game. There is a fat right tail where Conservative Player often makes $200, $400, $600 or even $800.00 on average per game. This pushes the mean of the histogram to the right:

2-player conservative game mean: $104.00/game

3-player conservative game mean: $241.00/game

4-player conservative game mean: $330.00/game

5-player conservative game mean: $369.00/game

6-player conservative game mean: $246.00/game

Of course poker is a zero-sum game, so the always call players opposing the Conservative players exhibit similar levels of variability and a mean loss of between $36.00 to $136.00 per game. Of worthy note is that in 2-player games the always call player rarely exceeds $0/game on average, while 3,4,5 and especially 6-player variants show the always call player occasionally making over $400.00 on average per

game. Our conjecture is that as the number of players increases, the Conservative player's reliance on 2-player Monte Carlo simulation causes his winning odds to deteriorate since the probabilities of winning in a two-player game are different then a 3,4,5 or even 6 player game. The two-player Monte Carlo simulation is optimistic.

The Smart player scenarios, shown in the bottom 5 rows exhibit a completely different pattern relative to the Conservative player. The Smart player's game mean is always above $0 and shifts outwards towards $200/game as more players are added. Smart player gains exhibit little variability compared to Conservative player. In fact, the highest variance for Smart players (6-player games) is about 2.5 times less than the lowest variant scenario for Conservative players (2-player games). Smart players consistently win at all tables, but never makes outsized gains like Conservative players do. Looking at the average gain over 100 games shows that Smart player does not lose any money on a per game basis at any table whereas a significant group of Conservative players do. On average, Conservative players still make more money in all scenarios than their Smart player counterparts. The mean game balances can be seen below:

2-player smart game mean: $70.00

3-player smart game mean: $124.00

4-player smart game mean: $146.00

5-player smart game mean: $197.00

6-player smart game mean: $206.33

According to the above information, Smart Player makes small consistent gains as he plays more tables. Conservative player often plays at a loss, but when he bets and succeeds it has a huge impact on his balance. This might be indicative of a player that makes large bets on very good card sets, but wins less frequently in other circumstances.

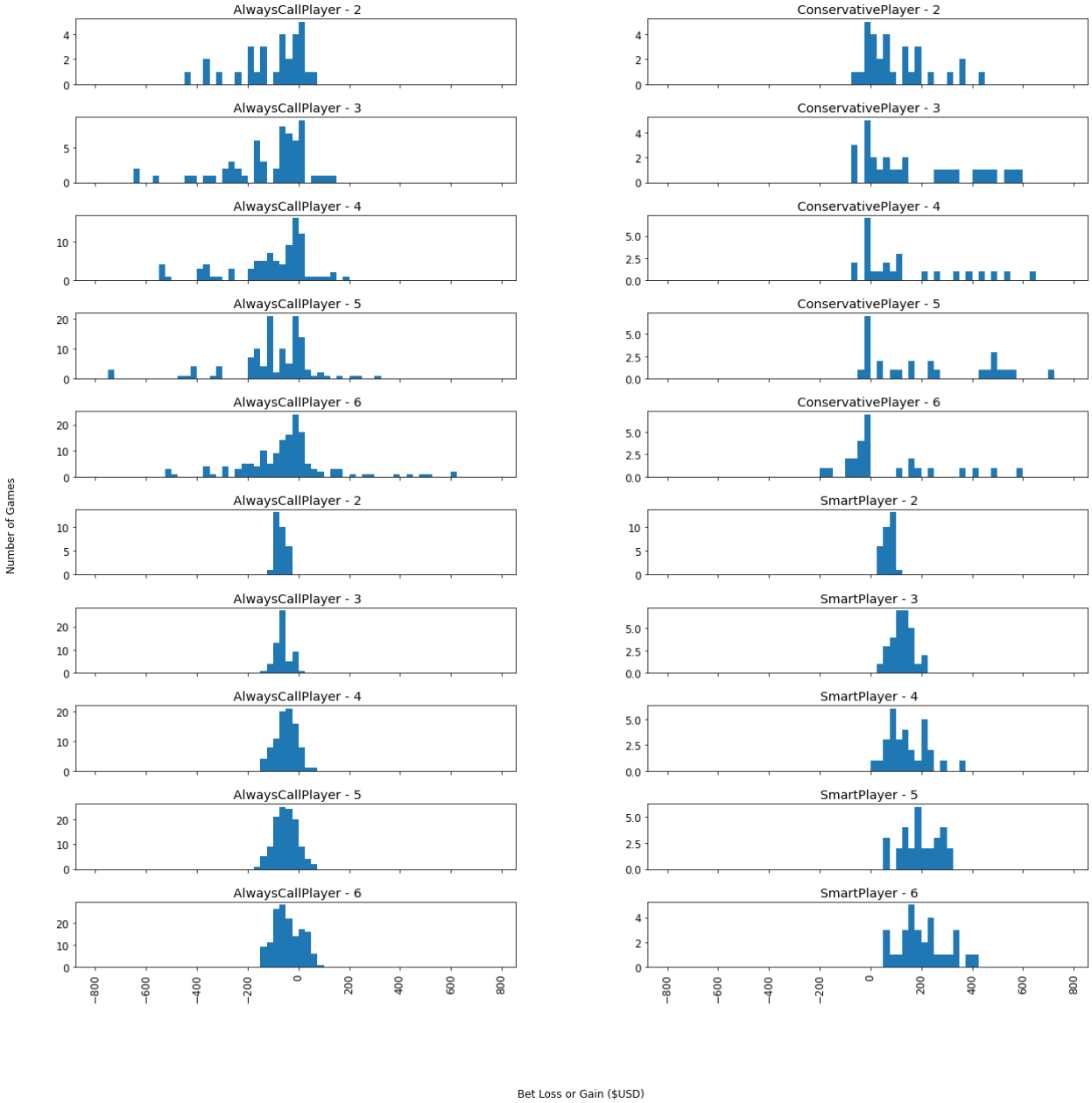

Fig.4: Bet Loss or Gain(USD) Histogram

Poker end game balance over many games:

After each game completes, we record the player's balance. We call this the end game balance. Studying the end game balance shows how player's gains or losses accumulate as he plays more poker hands. A good strategy will generally lead to players with an upward trending balance, while bad

strategies will have a downward trending balance. Before we plot the end game balance as a time series, it's worth looking at the overall distribution of end game balances for each of the four player strategies used in the first 2 scenarios. It's worth noting $100,000 is a player's beginning balance.

Looking at Fig.5., we see trends that confirm previous observations. For conservative player scenarios, both player strategies (Always Call Player and Conservative Player) exhibit high variability in end game balances. Conservative Players lowest end game balance was $81,432.00 and highest was an astounding $334,524.00; a 3-fold increase compared to his initial balance. Conservative players always call player opponents had minimum balance of $25,620.00 with the greatest balance they achieved being $162,239.00. Also highly variable, but one with players that experienced significant losses. The Smart player's end game balance ranged from a low of $95,375.00 to $141,790.00. Significantly less variability compared to the Conservative Player. His always call player opponents likewise had a less variable balance: $84,355.00 and a high of $111,350.00.

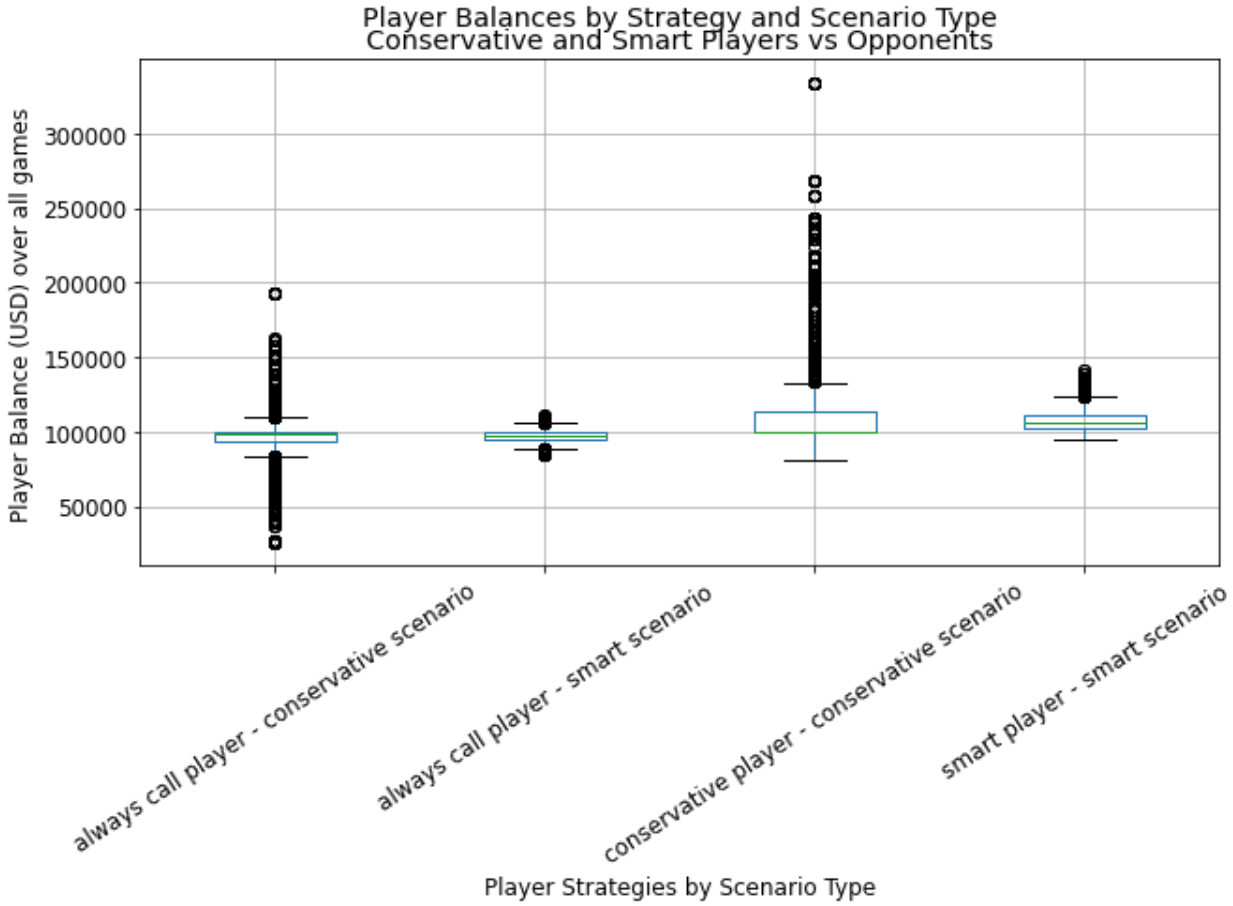

Fig.5. End game balance for each player by scenario type.

Adding a time-based component provides even more insight. In the following two graphs (Fig.6), the top row represents the Smart Player scenario, while the bottom represents the Conservative Player Scenario. The graphs depicts end game balance over a sequential series of poker games and the 95th confidence interval for that mean.

As you can see, both the Smart and Conservative players have an increasing trend in their mean end game balance, but the Conservative Players trends upwards at a significantly faster rate than Smart Players. The confidence interval is also wider for Conservative players hinting at increased variability and uncertainty. The most important observation is that Conservative players trend line compensates for the broader variability explaining why it has a higher average end game balance as time progresses. We believe the additional Always Call players is the main contributing factor to narrow confidence interval seen by Conservative Players always call player opponent. The Always call players had a lower loss rate and tighter confidence intervals in the Smart Player scenario when compared to the Conservative Player scenario.

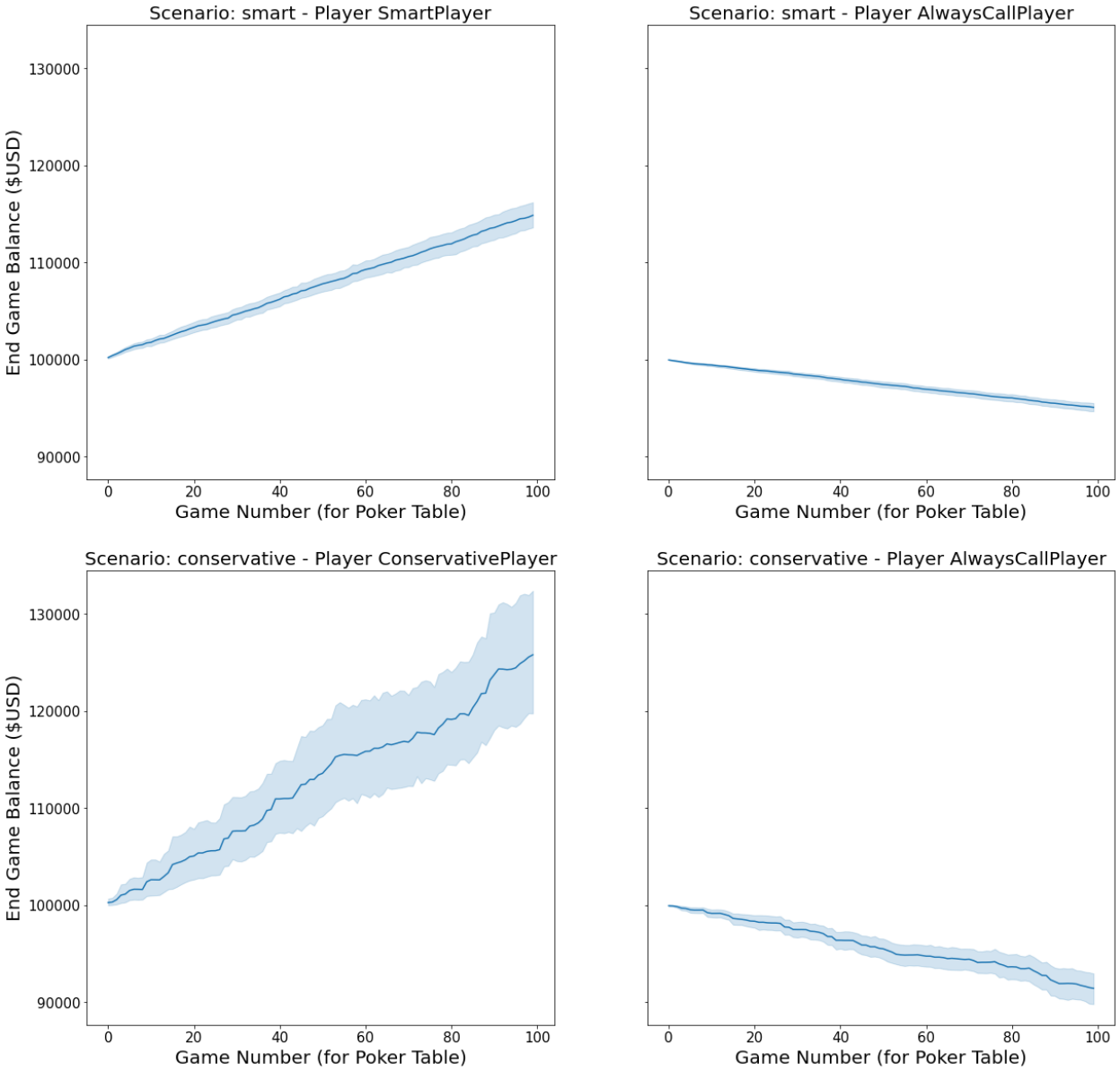

Fig.6. Strategy balance over number of games.

To explore the impact of number of players on the end game balance over time for the two scenarios, we tracked each player's balance over the number of games(Fig.7). In the above graph, the first two rows correspond to the Conservative Player scenarios, while the last two rows correspond to the Smart Player Scenarios. The Always Call Player graphs are shown on odd rows, while the Conservative and Smart players are shown on even rows. Number of players increases in left-to-right order from 2 players to 6 players.

The most interesting observation is that both Smart and Conservative Player mean end game balances trends upwards at a faster rate and the confidence interval expands as you add more players to the table. Our conjecture is that as the number of players increase, the average pot size and variability increases. Since both Conservative and Smart players often beat their Always Call player adversaries, they tend to take a larger proportion of the pot.

Another interesting observation is that for both Smart and Conservative Player scenarios, the Always Call Player lost less as you added more players to the table. Our conjecture is that when Conservative or Smart player faces a single adversary, that adversary has a single chance per game to beat the Conservative or Smart player counterparts. As you add more Always Call players, the odds of one of them beating the Smart or Conservative player increases diluting their average loss. Since always call players employ the same strategy, they should on average beat each other at the same rate. This explains why on average they lost games, but the average loss was lower when more Always call players were present.

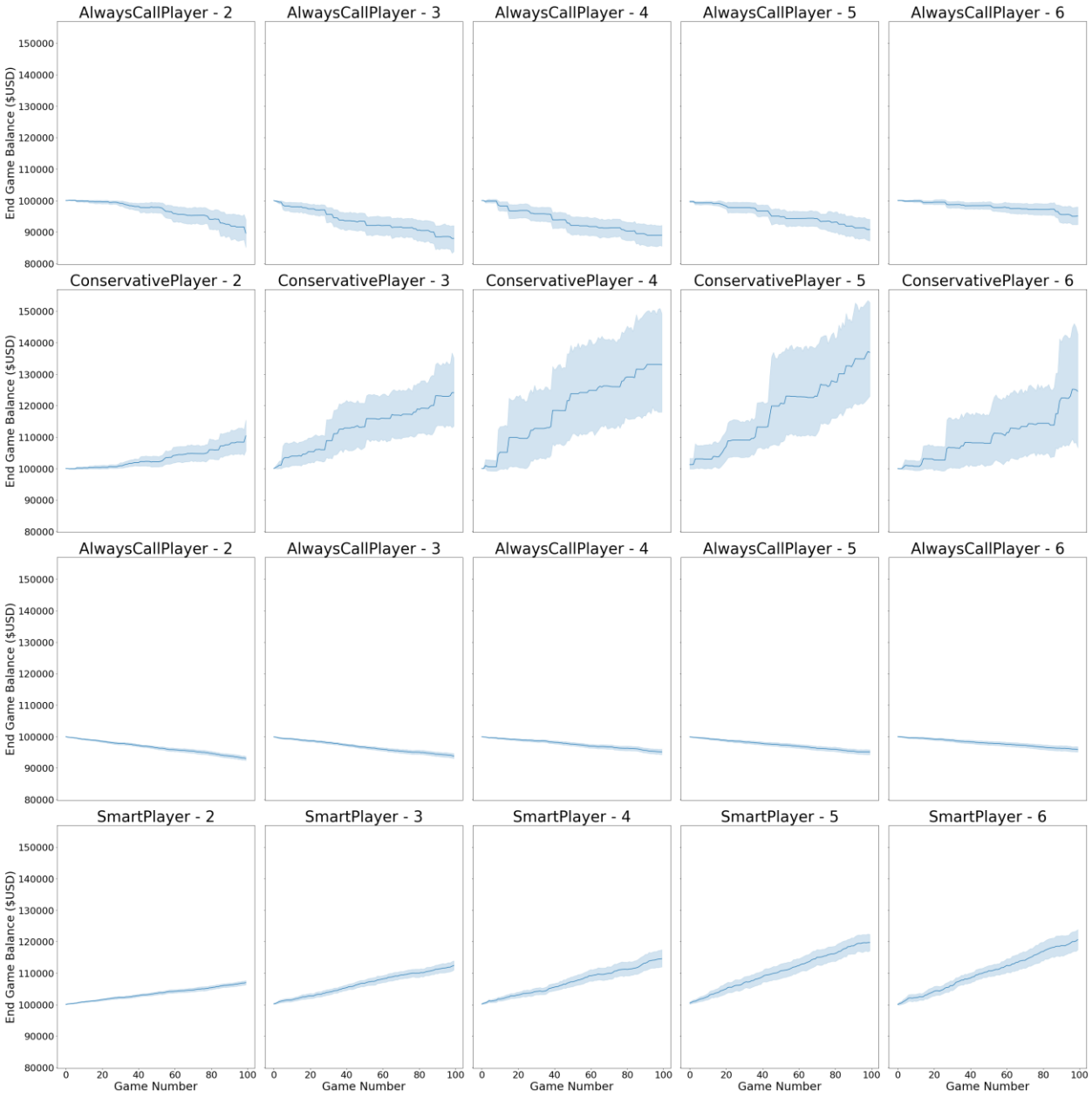

Fig.7. Player balance

Pre-flop odds compared to end game win and fold rates:

Looking at end game balance and per game gain/loss means is a good way of determining which player strategy is better. However, It does not tell us why those player strategies are superior and what one can learn from these strategies to improve ones own playing style. This section delves less into "what" happened and more into "how" the players came to their specific conclusions.

In this section, we will look at the 2-cards dealt to each player and then look at what percent of those hands went into one of four end game states: fold, last man standing, lost game and won game states. The end game states are as follows:

- **Fold:** the player folded his hand and loses the game.
- **Last man standing:** the player was the only individual that did not fold. He automatically won the round and didn't have to reveal his hand.
- **Lost Game:** the player did not fold during the game, had his hand scored and ended up losing. Hand scoring follows typical poker ranks: high card, pair, two pair etc.
- **Won Game:** the player did not fold during the game, had his hand scored and ended up winning/tying the round.

In the following series of heatmaps (fig.8 & 9), high card means the higher ranked card of the two-cards a player was dealt. Low card represents the lower of two cards dealt. We also use the term "same suite" meaning that both high and low card have the same suite: hearts, spades, clubs and diamonds as well as the "opposing suite" meaning the suites did not match. A pair in this analysis means both high and low card have the same rank an example being King-King, while a non-pair indicates both ranks were different. An example is King-Queen.

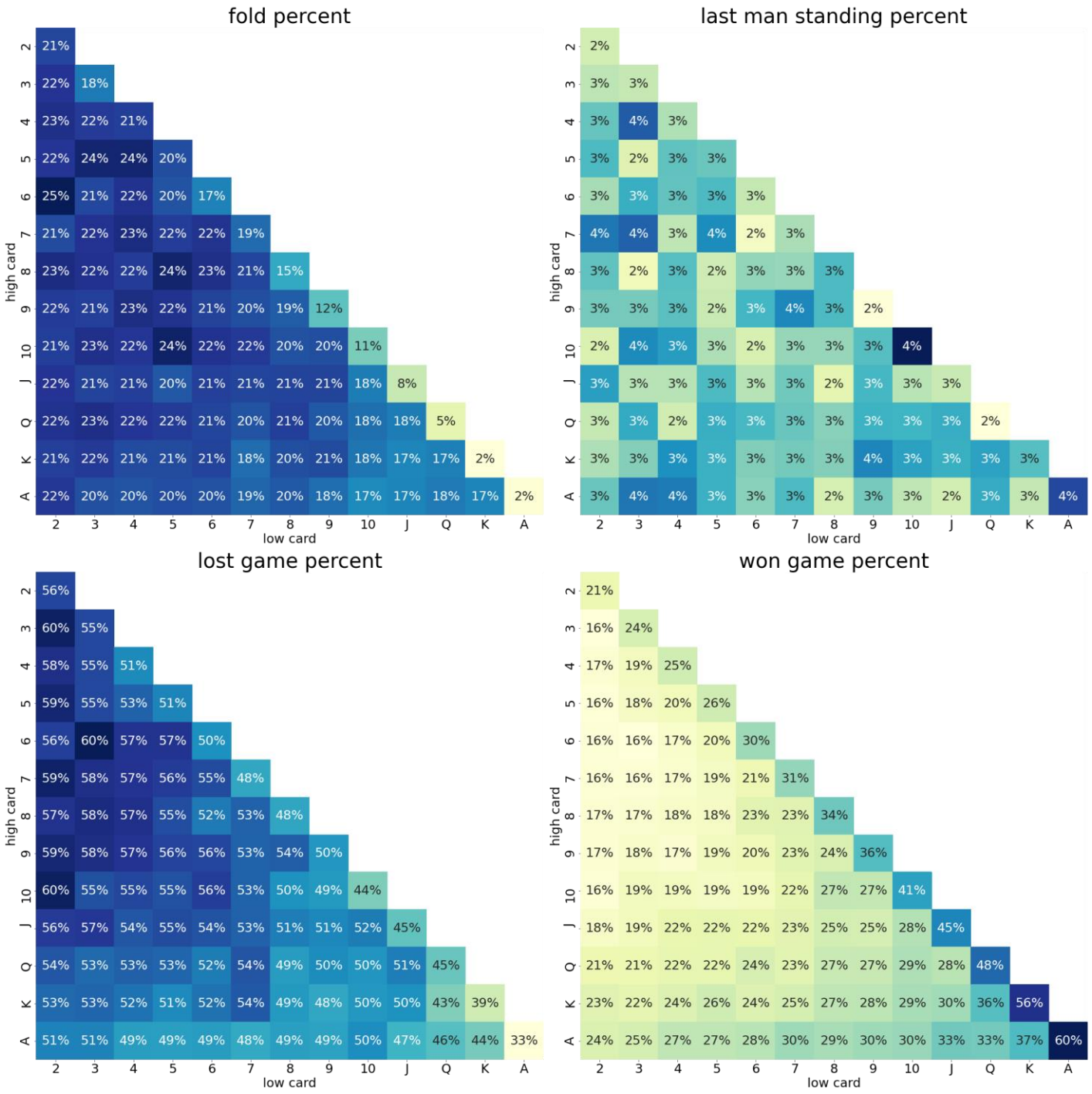

Fig.8. End game probabilities based on player's 2 card hand.

The above set of 4 heat maps (Fig.8) shows the end game states across all player strategies for both the Conservative and Smart player scenarios. This gives us a sense of the "typical" result for any given 2-card hand.

Some noteworthy observations about these results. The percent of 2-card hands that lost games is highly correlated with folding rates. They are inversely correlated with the winning game state. The last man standing state seems independent of the initial two-cards the player held. This should make sense to a seasoned poker player. The hands that lose tend to be the same one's people fold, while the hands that have good odds of winning tend to get folded less often. The randomness of the last man standing state also makes sense since most people fold their hands without knowledge of their opponents' hand. A more surprising observation is that pairs have higher win rates than non-pairs, lose less and are folded less often. The dominance of pairs becomes more significant as the rank of the pair increases with 2-pair winning only 21% of the time compared to 2-3 16% win rate, while an Ace-pair wins 60% of the time dropping dramatically to 37% with a A-K.

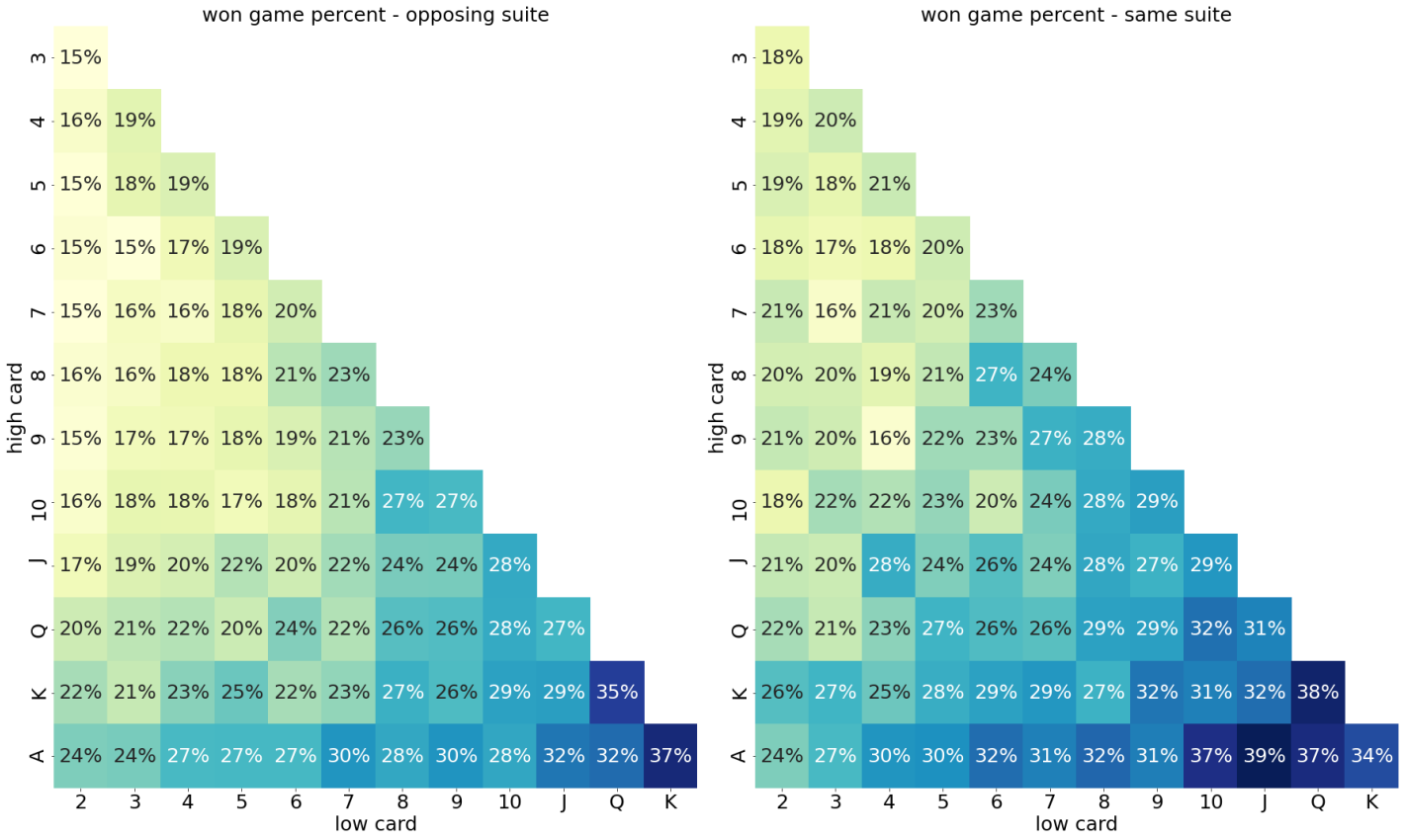

Fig.9. Percent game won based on first two cards.

Before we get into the analysis of end game results for different player strategies, it's worth looking at the difference between same suite and opposing suite non-pairs (pairs are always off-suite). In the above graph (Fig.9), we see won game rates for opposing suites on the left and same suites on the right. On average, having the same suite seems to increase win rates by approximately: 2.8%. This is less impactful than the authors expected. The benefit between pairs and their nearest off-suite neighbors was between 6% to 23%. Pairs seem to provide a greater benefit than having the same suite.

For the next three charts (Fig.10, 11 &12), we broke down the 2-card heatmaps by player strategy with always call player being in the first column, Smart player 2nd column and Conservative player 3rd column.

Rows represent three types of 2-card hands: same suite non-pairs 1st row, opposing suite non-pairs 2nd row and pairs 3rd row. We provide three series of 9 charts, the first for the game won state followed by the lost game state and the final set of 9 heatmaps for folded game state.

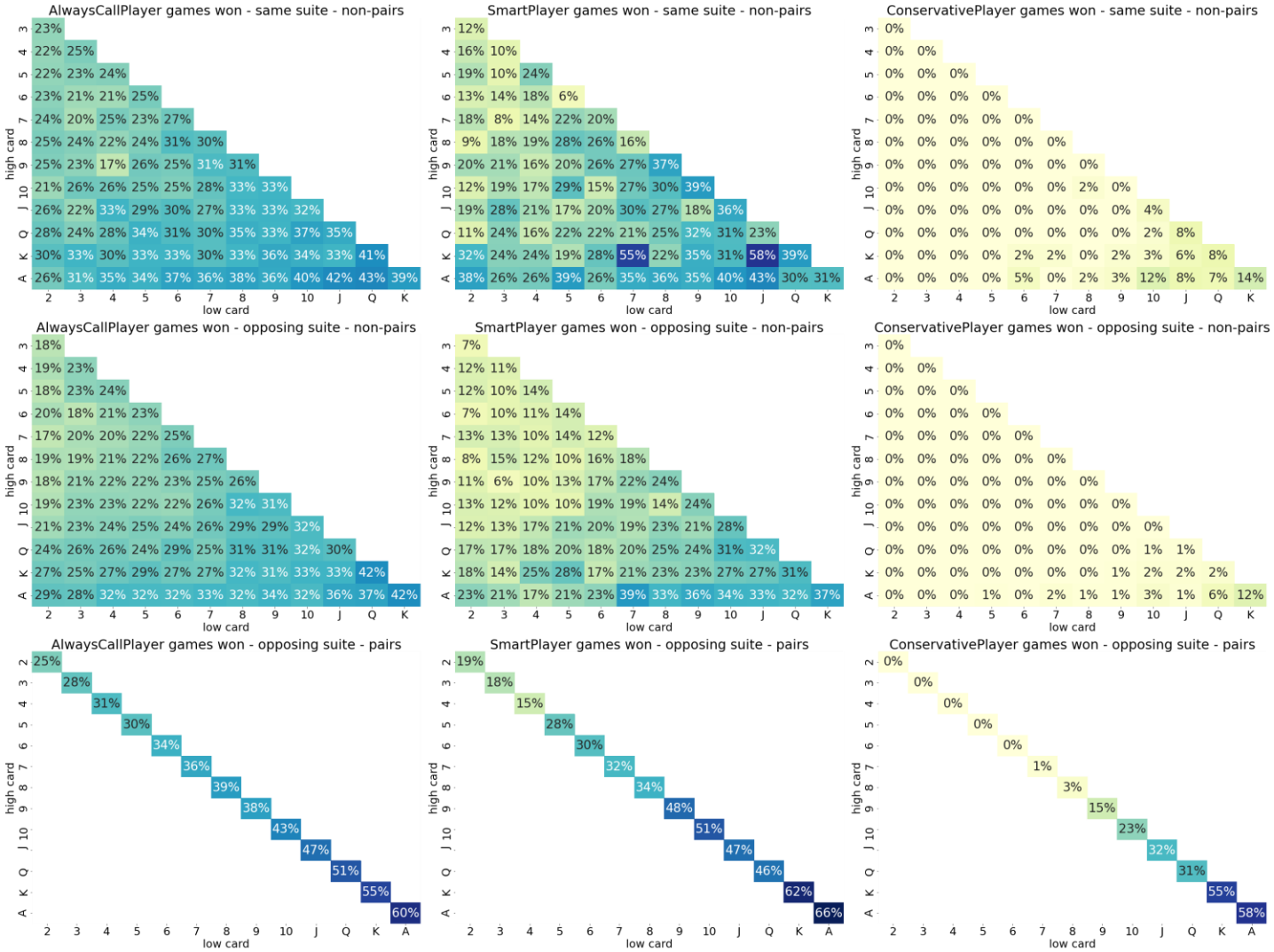

Fig.10. Probabilities of winning hands based on first two cards.

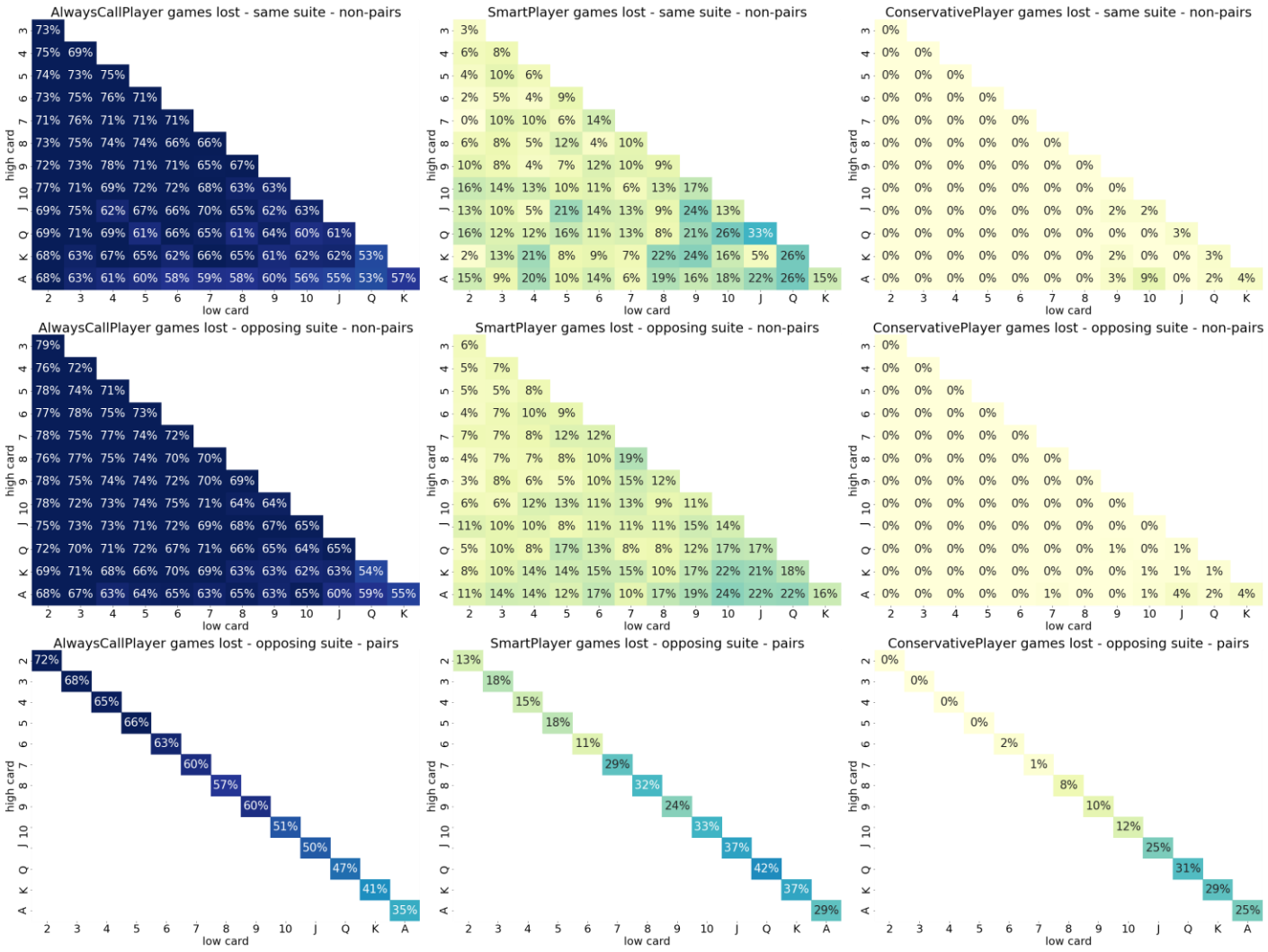

Fig.11. Probabilities of losing hand based on the first two cards.

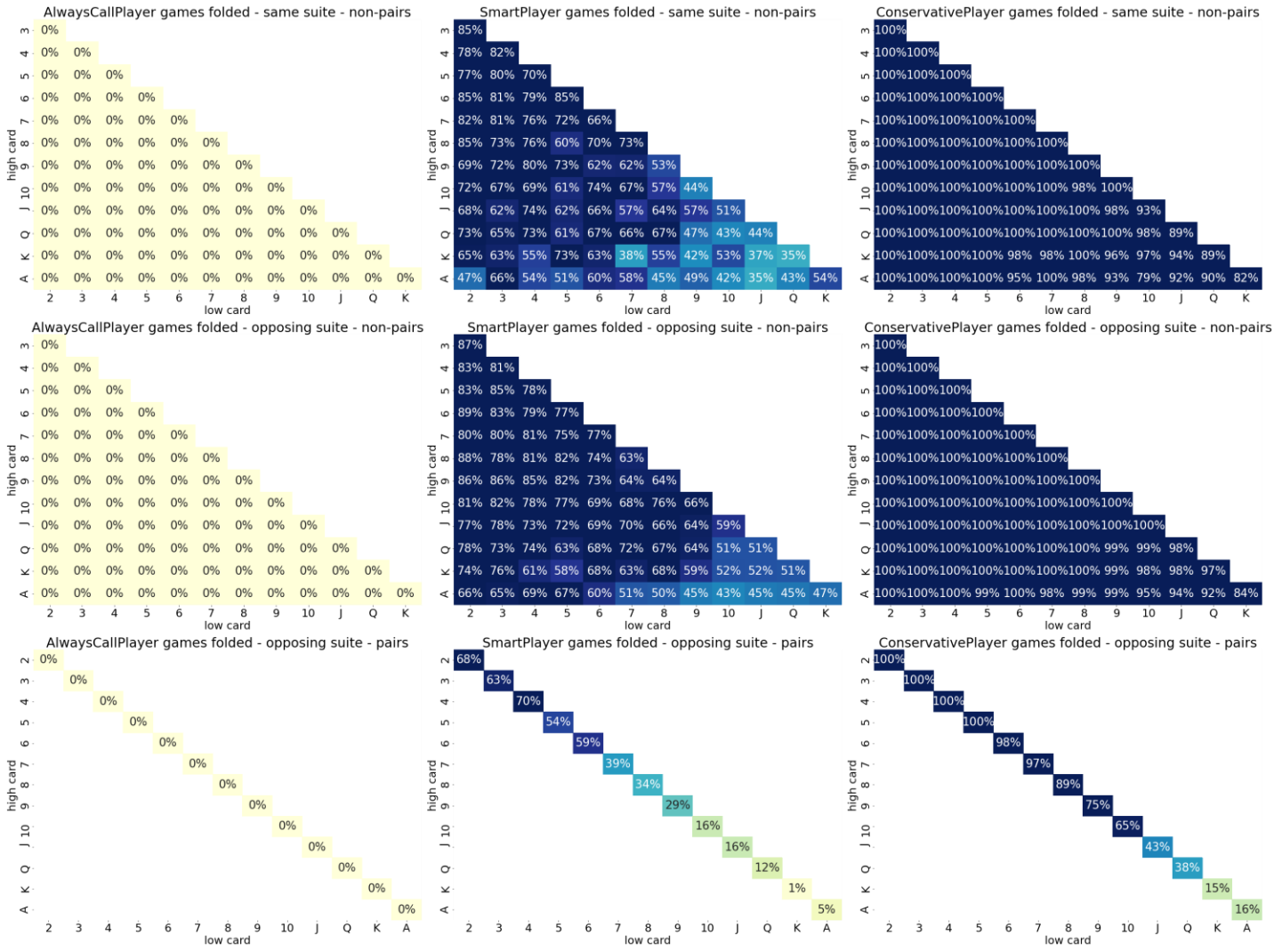

Fig.12. Probabilities of folding hand based on the first two cards.

These graphs provide an extensive amount of information when used in conjunction with each other. The most striking player strategy is Conservative Player. We can see that Conservative Player lost about as many games as he won since he plays so few hands. Conservative player plays almost exclusively 9-pairs or higher playing 9-pairs 25% and Ace-pairs 84% of the time. He tends to win these hands frequently. For non-pairs, he has a distinct preference for face cards and Aces: A-K, A-Q, A-J etc. Amongst these non-pairs, he prefers those with the same suite, but this is only a slight preference. He plays these cards less than 10% of the time. He never plays non-pairs where the high card is 9 folding these 100% of the time. On few occasions he tries K or Q high card. Conservative player is extremely selective for a poker player. He tends to wait for the perfect hand and then aggressively bets on it. This strategy works exceptionally well against the always call players who calls even his most aggressive bets and can't make significant gains when he folds. This explains the extreme variance observed since this player rarely wins a hand, but when he does he bets aggressively and almost always wins due to picking the best hands.

Let's contrast this with Smart Player. Smart Players win rates look astonishingly like the Always Call Player. In fact, he seems to win less games across almost all two-card combinations. The key difference in win rates seems to be that he wins non-face non-pair cards at significantly lower rates than the Always call player, while winning non-pairs with a high card of 10 or greater about as frequently. He performs about as well against the always call players with pairs. This difference can be explained by looking at Smart Players folding strategy. Smart Player has a distinct preference for face and ace cards especially when seen together folding those less than 50% of the time. He folds cards with non-face or ace cards between 60-85% of the time. Folds count as losses explaining the difference in the won game state. Smart player also favors 7-pair or higher folding 7-pairs only 39% of the time and Aces only 5%. Smart player compared to Conservative player is more open minded about his 2-card selections. He has won games with every single 2-card combination, but like Conservative player he has a distinct preference for face and ace cards. Like Conservative player he loves pairs especially high pairs. Smart player frequently folds his hands. It's just not nearly as often as Conservative player. The best way to visualize Smart Player is as a player that accepts his opening hand and then frequently adjusts his probability of winning in the 3, 4 and 5 community card phases. An interesting anomaly we noticed was that Smart player uncharacteristically loses same suited Q-J 43% more often than winning (loses 33% vs winning 23%). Smart Player might be getting the odds wrong (Fig.11 & 12).

Always call player never folds his hands. Hence his fold chart is 0% across the board. Always call player wins a lot of non-face card non-pairs independent of suite. This makes sense since Smart and Conservative players typically fold those hands. They also do surprisingly well at higher suites, potentially because Conservative and Smart player still fold a significant amounts of face and ace card non-pairs. Since every game has to have a winner those folded games must have been won by Always call players. This also explains why the Always Call Player has better 2-pair win odds overall. One note, since the always call player never raises, he never exploits those wins. One thing worth noticing is that the Always Call Player reaches the lost game state at much higher rates than the other two strategies. This makes sense since they Always Call Players never fold and so terrible hands that any other player would typically fold most frequently become losses for them.

The analysis focus is mostly on the differences in Conservative and Smart player strategies, so it's worth contrasting them quickly here. Conservative player plays almost no hands, but when he does play a hand he only plays the best and quickly escalates with raises. Smart player plays more broadly, but becomes limited by his fixed betting strategy meaning he never exploits his wins.

Conclusion

Based purely on mean number of chips won, Conservative player would be considered a better strategy to play. The reason is that, while Conservative Player can occasionally lose money at a 100-game poker table, he tends to only bet on the most powerful 2-card combinations and he bets a significant amount of chips during the process. Typically: 1%, 2%, 5% or 10% based on his probabilities of winning. Since he only bets when his win odds are typically above 75%, he wins almost all of these 2-card combinations and reaps a giant increase in his balance. On rare occasions, he will lose one of these combinations explaining why in a few tables he has a net loss on his average game. He doesn't bet frequently enough to offset these few bad losses at least not in the short term. However, in the long term so long as powerful 2-card combinations appear his balance skyrockets. A human player would counter this strategy by realizing that Conservative Player almost always plays pairs and should therefore fold every time Conservative Player decides not to fold. This would be considered the best counter play to the Conservative player. It's also hard to win against conservative player, because he won't bet outside really great 2-card hands. So this strategy will only force him out through attrition via collecting his small and large blinds.

Smart player definitely skews his play to more powerful 2-card hands, but will occasionally play weaker hands including 2-3 pair off suite. It seems that Smart player basis his calculations more on the community cards, which explains some of the folding behavior. For example, if Smart player gets a 6-5, he might play it due to having about equal odds of winning. When the first 3-community cards flips he readjusts his odds. If they are great, say a 6-5-4 indicating a double pair, he might increase his bet by $100.00, if the community cards make him worse off he might fold. This indicates a higher tolerance for 2-card pairs and more emphasis on the resulting post-flop phase. This tolerance can also be seen in the percent of wins required to proceed. Conservative player has a 75% bar that has to be cleared during each Monte Carlo simulation, while Smart player adjusts it based on the number of opponents he is facing. This results in Smart player winning on average lots of games, but due to only betting in fixed increments he can't take advantage of those wins. Hence Smart player loses out to Conservative player, because he has a non-aggressive strategy when it comes to the amount of money he bets and less so in terms of how he plays his cards and wins. Human players would find Smart player hard to counter. He plays weak 2-card pairs and strong 2-card pairs. He just plays the later more frequently. He tends to only bet on better hands, but this could happen in any stage of the game: pre-flop, 3, 4 or 5 card community. The only weakness might be that he signals a good hand when he bets $100.00 and he's right about that hand 75% of the time.

A good follow up strategy would be to merge the strengths of both players. Smart player should scale his bets based on his balance so that his small fixed increments don't slow down his gains over multiple games. Conservative player when playing against Always Call Players has a significant advantage over Smart Player, but that is due to his opponents never folding. A player that folds more frequently will cut Conservative Players gains quickly and potentially turn the match into a game of attrition by always calling when Conservative player folds and folding when Conservative player acts aggressively. This slowly transfers Conservative Players balance to the other players through the small and big blind mechanic (testable in Decima). The solution to this is to have Conservative Players occasionally bluff thereby forcing the other players to fold or play a complete round so that he doesn't lose his blind.

To check the performance of conservative player against smart player, we performed a quick simulation including all the strategies (Smart Player, Always Call Player, Always Raise Player, Calculated Player, Gamble by Probability Player). Surprisingly Conservative player outperformed Smart Player by the order of magnitude, which explains the strategic importance of how much you bet (Table.2). Conservative Player bets based on a percent of his current balance, while Smart player bets in fixed increments. The larger size of Conservative player's bet greatly assisted his balance's upward trajectory, while Smart players frequent wins did not compensate for his small bets. Another interesting finding was that Gamble by Probability strategy performs about as well as Smart Player. This makes sense since both players use a fixed increment betting strategy and tend to bet on the same cards. What is surprising is that Smart Player won significantly more hands than Gamble by Probability player (table.2&3). The last comparison we will make is with Gamble by Probability player and Calculated Player. Both won and lost the same number of hands. Calculated Player had a significant average loss of: $2,699. Gamble by Probability Player by contrast had a gain of $5,285. The reason the average gain or loss is interesting is that the cards they gambled on do not vary instead the amount they bet does. Calculated player calls any hand he expects to win and folds all other hands. Gamble by Probability player likewise folds any hand he expects not to win, but unlike Calculated player raises the bet by his percent chance of winning eg 50% win chance results in a $50 bet. That means that the difference in average gains or losses is due purely to the amount gambled and not on the cards they gambled on. A theme that is echoed in the Conservative vs Smart player matchup.

Strategy	Average gain/loss
Always Call Player	-$26,087
Always Raise Player	-$23,099
Calculated Player	-$2,699
Gamble by Probability Player	$5,285
Smart Player	$5,300
Conservative Player	$41,299

Table.2. Average gain/loss for all players.

Strategy	Total Number of Game Won	Total Number of Game Lost
Conservative Player	34	2966
Calculated Player	263	2737
Gamble by Probability Player	263	2737
Smart Player	610	2390
Always Raise Player	978	1899
Always Call Player	995	1882

Table.3. Total number of game won/lost for each strategy.

Group members: Christopher Kottmyer, Shahin Shirazi

References:

[1] https://en.wikipedia.org/wiki/Poker_probability