Math 381 Group 3:
Monte Carlo Simulation of the Card Game *Durak*

Authors:
Angela An
Emma Liao
Zach Shaw
Kyle Zembroski

Table Of Contents

Introduction

In our study, we used the Monte Carlo method to model a card game called *Durak*. The aim of our project is to find the best winning strategy of attacking and defending among the four strategies we have set. The four strategies were chosen by each one of our group members based on our experiences. For our project, we have decided to focus on modeling *Durak* with four initial players. *Durak* is a card game that starts with a deck of 36 cards, which consists of cards of values 6 to Ace of all suits from the full 52-card deck. The Jack, Queen, King and Ace cards take on the values 11 to 14 accordingly.

After the cards have been dealt, a player is randomly picked to start off the game. This person gets to be the first attacking player. The attacking player attacks the person sitting next to him or her in the circle. The players are seated in a circle. The next player then has to defend the cards placed by the attacker with cards that have a higher value of the same suit as each of the attacking cards. Any player that is not defending is also allowed to attack the defender by placing cards of the same rank as those that are on the table. The defender then has to defend all of these cards. If a player places a card on the table, the attacking players are once again allowed to attack with cards of the same rank. The defender only gets a chance to attack the next player if he successfully defends the attackers' cards. If he fails to defend every card, he loses his chance to attack the player after him. The defender discards the cards he used to defend the attacking cards plus the attacking cards to the discard pile if he is able to defend successfully. If he is unsuccessful, he adds his defending cards and every attack card to his hand.

For our model, we ranked the cards numerically, 6 through 14, and players obtain the attack cards and defense cards based off of the strategies we defined. Each player is assigned an attack and a defend strategy. We then start testing these strategies to determine which ones give the best chance of winning. Winning has a different definition in this game as compared to other games. In this game, winning is defined as running out of cards. In a four person game of *Durak,* there are three winners and one loser. The first three people who run out of cards are considered "safe" and are removed from the game. The last player with cards in his or her hand is considered the loser. We collected the number of losses for each player. The win rate for each player is then the total number of losses per the number of games played.

Background

Inspiration

We all enjoy playing card games in our spare time, so we had a discussion about what card games each one of us played. We had several candidates such as *Texas Hold 'em, Blackjack,* and *Durak*. We noticed that the difficulty level of *Blackjack* is relatively easy and there is not much variation to it. In other words, the best strategies are relatively intuitive for the players to choose from.

On the other hand, *Texas Hold 'em* can be challenging to model because there are lots of considerations. For example, constructing a model to maximize the revenue earned in a game of *Texas Hold 'em* would be difficult because the model would have to estimate the strength of opposing player's hand based off of the cards that the dealer has flipped, and the actions the other players have made in the round. These actions include: how they bet and body language. It can be tricky to model the game if we take all of these considerations into account. Since we had limited time to complete this project and we really want to find the best strategies for a particular card game to the greatest extent, we settled on *Durak. Durak* has a moderate level of complexity that grants flexibility with modeling, while still being feasible to model.

History and Previous Works

There are many applications of the Monte Carlo simulation. First we found that a group of students from Stanford had conducted a similar study on the playing strategies for *Durak*. Compared to this study, we used the Monte Carlo method to test strategies that a human could implement in a real game of *Durak*; whereas they used reinforcement learning techniques and the temporal difference learning framework to find the general best strategy.[2] We designed two attack and two defense strategies based off of playing the game. The goal of these strategies is to minimize the chance of losing. In their study, each state of game will be the input data and their trained model can predict the chance of winning for different actions (playing a card, taking, and passing) for each turn. Every win will give the model insight to the success of the selected actions while every loss will not. In other words, they constructed an Artificial Intelligence agent for the card game *Durak* to play the best strategy based on the trained model.

The first person who modeled a card game using Monte Carlo methods is Stanislaw Ulam. In 1940s, while he was playing *Solitaire*, he created the idea of random

experiments. He developed the idea of random experiments into the Monte Carlo method.[1] The name Monte Carlo came from Stanislaw's uncle gambling at the casino of the same name. [1] At that time, the Monte Carlo method was key in his research to solve the problem of neutron diffusion in fissionable material. In his study, he also used Markov Chains and matrix theory to perform a finite number of experiments to estimate the probability of producing a given particle within certain conditions. Both our and Ulam's research used the Monte Carlo method to estimate the probability of a phenomenon. However, we applied similar techniques to two different applications. His research focused on solving physical problems. In our study, the Monte Carlo method was used for estimating the winning rate of each strategy, and therefore to find the best strategy to not lose at *Durak*.

The Model

In our project, we used the Monte Carlo method to model a card game called Durak. The aim of our project was to determine the best attack and defense strategies to increase one's chances of winning the game. As a group of four, we decided to model the game to include four players in total so that we could draw insights from playing the game amongst ourselves. By playing the game ourselves, we came up with a couple of feasible strategies a person playing *Durak* would use to try to win the game. We then gave each player a different combination of an attack strategy and a defense strategy we came up with.

We set up the game in a way such that each player would begin with a completely random set of cards each round. We did this by shuffling the deck of cards and then dealing the top card of the deck to each player in a round according to how the players are seated. To keep this as close to how the game is played in real life, the players (each with their assigned combination of attack and defense strategies) are randomly assigned a seat in the circle and a random player is picked to start off the game.

Data

In order to determine viable strategies to test, we played *Durak* ourselves and took note of the strategies each of us employed throughout playing the game. We found that each one of our group members tried to save their high cards for later in the game. This led to our strategies of attacking and defending with low value cards. Upon digging deeper into the strategies, we uncovered some more. Some group members picked up on the importance of high value cards, and started picking up high cards when they were attacked. This led to one of our extensions. One of the subtler strategies was defending

with trump cards. This strategy was important to remove high cards from play. Ultimately, we settled on 2 attack strategies and 2 defense strategies. Our attack strategies are: **attack with high cards** and **attack with low cards**. Our defense strategies are: **defend with low cards** and **defend with trump cards**.

One interesting thing that we discussed is the fact that our strategies changed as the game progressed. We found that we were more likely to take cards earlier in the game; and as the game progressed, which is tracked by the draw deck reducing in size, we were more likely to defend cards than take them into our hand.

Implementation

To find out exactly which strategies increases one's chances of winning, we came up with the following combinations of strategies:

- 1 player: Attack with high cards & defend with trump cards
- 1 player: Attack with low cards & defend with low cards
- 1 player: Attack with low cards & defend with trump cards

In each of these combinations, the other 3 players would have the following strategies:

- 3 players: Attack with high cards & defend with low cards

We then ran these individual combinations 1,000 times and recorded the probability that each player won in each round. We also conducted 45 such trials to get the results in the Data Tables section. Each trial consisted of 1,000 games to give us what we felt was an accurate representation of the average win percentage.

Simplifications

In order to make our model feasible, we had to make some simplifications to the game that we are modeling. These simplifications are: the trump suit doesn't play a role in determining which card to attack with; the player that lost the current game doesn't get attacked first in the next game; if you have a card of equal value to the attacking card, the defender can not pass the defensive responsibility to the subsequent player. We also omitted adaptable strategies based off of where in the game we are. These simplifications resulted in gameplay that diverges slightly from the actual game. The two simplifications that create the largest divergence are the passing of the defensive responsibilities to the next player and adaptable strategies based on size of the draw deck.

Passing the attack drastically changes the way we track the incrementation of players. If both players fail the defense, then attacking skips both defenders. For example, if Player 1 attacks Player 2 with three 9's, and Player 2 plays one 9 and passes the attack to Player 3, Player 3 has to defend all four cards of rank 9. If Player 3 fails to defend the 9, Player 3 must pick up all the cards used in the attack. The round starts with Player 4 attacking Player 1. This changes the logic used to determine the next attacking player.

Adaptable strategies based on the size of the draw deck is something that we discovered while playing through the game ourselves. This would change our attack and defense strategies as each game progresses. Changing the strategies will change the attack cards and the defense cards.

Results

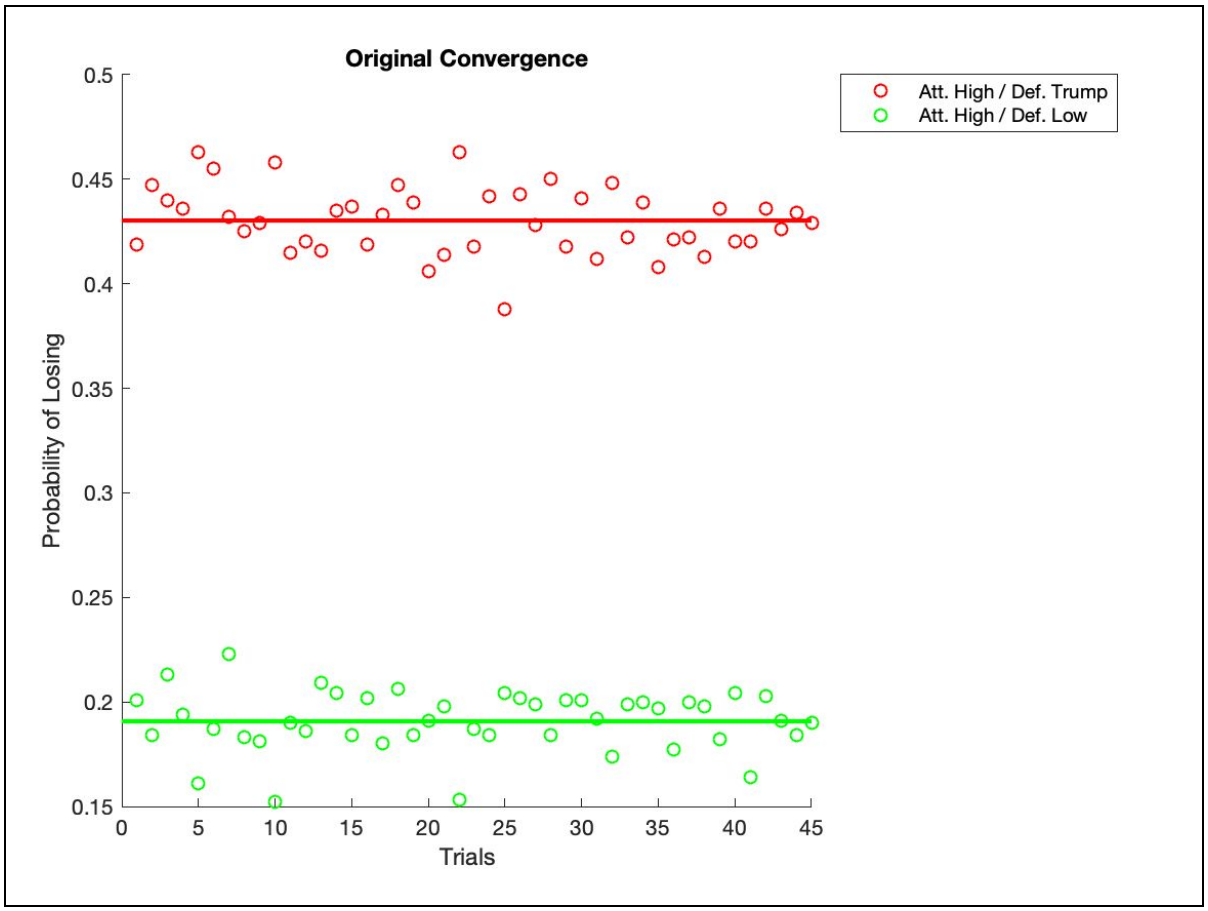
We ran the simulation for 1,000 games and collected data for 45 trials. The original model examined four different strategies. As mentioned previously, we recorded the number of times a player with a certain attack and defend strategy would lose in 1,000 games. We then executed 45 such trials to get the average losing probability that is presented in the data tables.

The first strategy that we examined was one in which a player attacked with his/her highest cards and defended with trump cards. The other three players attacked with high cards and defended with low cards. We ran the simulation for 1,000 games and collected data for 45 trials. We are 95% confident that the mean chance of losing a game of *Durak* is 0.4303 ± 0.00462 for a player attacking with high cards and defending with trump cards. The players that attack with high cards and defend with low cards will lose about 19% of the time.

The second strategy that we examined was when one player attacked and defended with his/her lowest cards. From the data collected, we are 95% confident that the mean probability of a player losing a game by using this strategy is within the range of 0.1052 ± 0.00250. The other three players that attack with high cards and defend with low cards will lose about 30% of the time.

Lastly, we examined a strategy in which a player always attacked with his/her low cards and always defended with trump cards. We are 95% confident that the mean probability of this player losing a game falls within the range 0.2869 ± 0.00467. The other players that attack with high cards and defend with low cards had a mean losing probability of about 0.2371 ± 0.00416, 0.2393 ± 0.00379, and 0.2367 ± 0.00397.

This model successfully describes the success rate of different strategies used in a game of *Durak*. The margin of error is less than 0.5% for all strategies. Since the margin of error is low, 1000 games played per trial for 45 trials is a reasonable sample size. This allows us to use the Central Limit Theorem, and assume that the current data is modeled by a relatively normal distribution; this means as the number of trials increase, we expect the model to predict more reasonable losing percentages for each strategy. The variability between the losing probabilities of the other three players is likely due to the movement of the players between each game. The below graph shows the probability of losing against the number of trials. It clearly shows that each of the two strategies tested in each trial approaches an average loss probability. Each graph presented in the Appendix has 2 sets of data: data for the test strategy and data for one of the control strategies. All three of the control strategies ended up with a very similar probability. To simplify the graphs, only one control player is plotted. Plots for the other test cases and both of our extensions can be found in the Appendix: Plots (starting at pg. 21)



Extensions

Extension 1: Cannot Attack More Than Defender's Hand

One thing we encountered while testing the model was that we did not check the size of the defender's hand when determining how many cards to attack with. This was particularly noticeable when the model reached the endgame. Occasionally, people would attack the defender with more cards than they had in their hand. This resulted in forcing the defender to pick up all of the attack cards, even though they couldn't feasibly defend all of the attack cards. We analyzed the same combination of strategies that were in the original model.

We are 95% confident that player that attacks with high cards and defends with trump cards will lose with a probability of 0.3904 ± 0.00452. This strategy has a mean losing probability that is 11.63% less than that of the mean losing probability of the original model. The three players that play against this strategy have a losing probability of about 20%.

There is a 95% chance that a player using a strategy that always attacks and defends with low cards will lose with a probability of 0.1674 ± 0.00316.

A player that attacks with low cards and defends with trump cards will lose a game of *Durak* with a probability of 0.3237 ± 0.00481 95% of the time. The three other players that attack with high cards and defend with low cards will lose about 22% of the time with the same confidence.

Extension 2: Defender Picks up High Value Cards

Our next extension was to have the defender pick up high value cards. This was a situation that arose in our initial playing of the game. This was something that changed as our own game progressed. Players within the simulation would pick up cards if the following three conditions were met:

1.) The deck is not empty
2.) The player is attacked with at most 2 attack cards
3.) The cards in the attack are either Kings or Aces.

Using this extension on the original combination of strategies, there is 95% chance that the strategy in which a player attacks with high cards and defends with trump cards will lose a game with a probability of 0.3140 ± 0.00400 when playing against three players that attack with high cards and defend with low cards. Each other player has a mean, losing probability of about 0.229.

Conclusions

Applying the Monte Carlo method to model *Durak* was extremely effective. This method allowed us to collect the average loss rate of each player, and suggested which strategy is the best.

It *seems that* the best strategy is to attack and defend with low cards in the original model and in the extensions. As seen in the data tables, the player that attacks and defends with the lowest possible cards seems to have the lowest mean probability of losing in a game of *Durak* against players that play with high value cards and defend with low value cards. The player that utilizes this combination of attack and defense strategies have a mean losing probability of around 0.1 and no higher than 0.167 in the original model and the extensions. This is a pretty good probability in a game played with four players.

Of course, we recognize that one can counter that attacking and defending with low cards seems to be the best combination of strategies but only when playing against three other players that all attack with high cards and defend with low cards.

Realistically speaking, attacking and defending with low cards seems to be a pretty optimal strategy in *Durak*. It makes sense to attack with low cards, since the player would be losing those cards regardless of whether the defender can defend it or not. It also makes sense to defend with low cards since the cards would be discarded if defended successfully or passed on to the next player if the same value card is used to defend. In other words, it would be unnecessary to defend with a very high value card like an Ace for an attack card rank of 6 when the defender had a card rank of 7. Attacking and defending with low cards was definitely a strategy that some of us implemented when we were playing *Durak* amongst ourselves and the results of our model seems to reflect that.

References

[1] Metropolis, N. “The Beginning of the Monte Carlo Method”. Los Alamos Special Issue, 1987. https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9067. Date of Access: March 6th, 2019.

[2] Nguyen,Sammy and Narek Tovmasyan. Stanford CS 229 Project Paper. http://cs229.stanford.edu/proj2013/Learning%20Game%20Playing%20Strategy%20for%20Durak_SN%20&%20NT.pdf. Date of Access: March 6th, 2019.

Appendix

Data Tables

Original:

	Att. high / Def. trump	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.430266667	0.190733333	0.188022222	0.190977778
Standard Deviation	0.015806213	0.014677441	0.012794688	0.010506467
Margin of Error	0.004618167	0.004288369	0.003738277	0.003069719
Confidence	0.4303 ± 0.00462	0.1907 ± 0.00429	0.1880 ± 0.00374	0.1910 ± 0.00307

	Att. low / Def. low	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.10517778	0.30051111	0.2976	0.29671111
Standard Deviation	0.00853945	0.01311769	0.01245793	0.01424401
Margin of Error	0.00249501	0.00383265	0.00363989	0.00416173

Confidence	0.1052 ± 0.00250	0.3005 ± 0.00383	0.2976 ± 0.00364	0.2967 ± 0.00416

	Att. low / Def. trump	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.28688889	0.23711111	0.23928889	0.23671111
Standard Deviation	0.01600032	0.01422262	0.01298009	0.01364588
Margin of Error	0.00467488	0.00415548	0.00379245	0.00398697
Confidence	0.2869 ± 0.00467	0.2371 ± 0.00416	0.2393 ± 0.00379	0.2367 ± 0.00397

Extension 1 (size of attack <= size of player hand)

	Att. high / Def. trump	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.39035556	0.20522222	0.20342222	0.201
Standard Deviation	0.01546867	0.01252069	0.01300785	0.01457738
Margin of Error	0.00451955	0.00365822	0.00380056	0.00425913
Confidence	0.3904 ± 0.00452	0.2052 ± 0.00366	0.2034 ± 0.00380	0.2010 ± 0.00426

	Att. low / Def. low	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.16742222	0.27793333	0.27977778	0.27486667
Standard Deviation	0.01082818	0.01330653	0.01111896	0.01485475
Margin of Error	0.00316372	0.00388782	0.00324867	0.00434018

Confidence	0.1674 ± 0.00316	0.2779 ± 0.00389	0.2798 ± 0.00325	0.2749 ± 0.00434

	Att. low / Def. trump	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.32368889	0.22777778	0.22666667	0.22186667
Standard Deviation	0.01647563	0.01385458	0.01244625	0.01174657
Margin of Error	0.00481375	0.00404795	0.00363647	0.00343204
Confidence	0.3237 ± 0.00481	0.2278 ± 0.00405	0.2267 ± 0.00364	0.2219 ± 0.00343

Extension 2 (defenders pick up cards if deck is empty, only two attack cards placed, and card ranks are greater than queen)

	Att. high / Def. trump	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.31395556	0.22946667	0.22744444	0.22913333
Standard Deviation	0.01368801	0.01082632	0.0163423	0.01391794
Margin of Error	0.00399928	0.00316317	0.0047748	0.00406646
Confidence	0.3140 ± 0.00400	0.2295 ± 0.00316	0.2274 ± 0.00477	0.2291 ± 0.00407

	Att. low / Def. low	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average:	0.150955556	0.282133333	0.2826	0.28431111

Standard Deviation	0.011579676	0.014853221	0.01450141	0.01542993
Margin of Error	0.003383282	0.004339728	0.00423694	0.00450823
Confidence	0.1510 ± 0.00338	0.2821 ± 0.00434	0.2826 ± 0.00424	0.2843 ± 0.00451

	Att. low / Def. trump	Att. high / Def. low	Att. high / Def. low	Att. high / Def. low
Average	0.201977778	0.26262222	0.2684	0.267
Standard Deviation	0.012944793	0.01134839	0.01155107	0.01364818
Margin of Error	0.003782134	0.00331571	0.00337493	0.00398764
Confidence	0.2020 ± 0.00378	0.2626 ± 0.00332	0.2684 ± 0.00337	0.267 ± 0.003988

Python Code

```
#Attack strategies: Highest available card, lowest available card (ignores trump).
#Defense strategies: Always attempt to defend each card.  Defend with trump and
defend with the lowest card.  Defenders may also pick up the attack cards if the
following conditions have been met:  1.) At most two attack cards have been placed
that are of rank King or Ace. 2.) The deck is not empty.
import random
import sys
import csv

DIAMONDS = 'D'
HEARTS = 'H'
CLUBS = 'C'
SPADES = 'S'

suits = [DIAMONDS, HEARTS, CLUBS, SPADES]

class Card:

    def __init__(self, suit, rank):
```

```
        self.suit = suit
        self.rank = rank


    # string representation of card "(suit, rank)""
    def __repr__(self):
        return "({}, {})".format(self.suit, self.rank)


    # cards can be compared based on rank.
    def __cmp__(self,other):
        if self.rank < other.rank:
            return -1
        elif self.rank > other.rank:
            return 1
        else:
            return 0

    # allows hashable cards (can add them to a set)
    def __hash__(self):
        return id(self)

class Player(object):

    def __init__(self, num, attack_strategy, defense_strategy):
        #Initialize player number
        self.num = num
        #Initialize hand
        self.hand = []
        self.attack_strategy = attack_strategy
        self.defense_strategy = defense_strategy

    def attack(self, size_of_defender_hand):
        cards = []
        size_of_attack = 0
        while size_of_attack < size_of_defender_hand and size_of_attack ==
len(cards):
            cards.append(self.attack_strategy(self.hand))
        self.remove_cards_from_hand(cards)
        return cards

    def defend(self, offense):
        cards = self.defense_strategy(offense, self.hand)
        self.remove_cards_from_hand(cards)
        return cards

    def remove_cards_from_hand(self, cards):
```

```
        for card in cards:
            self.hand.remove(card)

    #takes array to represent a single card.  Adds card to hand
    def add_to_hand(self, card):
        self.hand.append(card)

    #return true if 0 cards in hand, else return false
    def is_empty(self):
        return (len(self.hand) == 0)

    def __repr__(self):
        return "({})".format(self.num)


    # if cards are on the table, place cards that have the same rank.
    # Returns a card with same rank for that players attack strategy.
    # returns None if no card matches.
    def match_cards(self, card_on_table, attack, size_of_defender_hand):
        for card in attack:
            if card == card_on_table:
                # remove card from attack list, then add left over attack cards
back to the list
                attack.remove(card)
                self.hand.extend(attack)
                return card
        return None


class Game:

    def __init__(self, deck):
        self.players = []
        self.deck = deck

    def add_players(self):
        player_one = Player(1, attack_with_low, defend_with_low)
        self.players.append(player_one)
        player_two = Player(2, attack_with_high, defend_with_low)
        self.players.append(player_two)
        player_three = Player(3, attack_with_high, defend_with_low)
        self.players.append(player_three)
        player_four = Player(4, attack_with_high, defend_with_low)
        self.players.append(player_four)
        random.shuffle(self.players)
```

```
    # deal 6 cards to each player
    def deal(self):
        self.deck.shuffle()
        for i in range(24):
            card = self.deck.pop_from_deck()
            self.players[i % 4].add_to_hand(card)


    # returns false if there are more than one players in the game
    def game_over(self):
        if len(self.players) == 1:
            return True
        return False


    # Can only attack with as many cards as are in the defenders hand
    def play_round(self):

        if not(len(self.players) == 1):
            cards_on_table = []
            first_attacker = self.players[1]
            defender = self.players[0]

            size_of_defender_hand = len(defender.hand)
            attack_cards = first_attacker.attack(size_of_defender_hand)
            size_of_defender_hand = size_of_defender_hand - len(attack_cards)
            # create list of all attacking cards after attacker places cards

            attack_cards.extend(self.add_to_attack(attack_cards,
size_of_defender_hand))

            defending_cards = []

            # defender doesn't pick up cards if size of attack > 2 or a card is a Q
or below
            # also doesn't pick up cards if the deck is empty.
            if len(attack_cards) > 2 or attack_cards[0].rank <= 12 or
self.deck.size() > 0:

                # list of cards use to defend by defender
                defending_cards = defender.defend(attack_cards)


                # list of all cards on the table including attack & defending cards
                cards_on_table.extend(attack_cards)
                cards_on_table.extend(defending_cards)
```

```
                # while defender is still defending && attackers are still
attacking
                while len(defending_cards) == len(attack_cards) and
len(attack_cards) > 0:
                    size_of_defender_hand = len(defender.hand)
                    attack_cards = self.add_to_attack(defending_cards,
size_of_defender_hand)
                    defending_cards = defender.defend(attack_cards)
                    cards_on_table.extend(attack_cards)
                    cards_on_table.extend(defending_cards)


            # ordering of draw is incorrect
            if len(defending_cards) < len(attack_cards) or defending_cards is None:
                defender.hand.extend(cards_on_table)
                self.players.insert(0, self.players.pop())

            self.players.insert(0, self.players.pop())
            self.check_player_status()
            self.draw_cards()

    # all attackers attack defender based on the size of his/her hand
    def add_to_attack(self, cards_to_match, size_of_defender_hand):
        # if defender has more cards in hand than those that were attacked with
        # each player attacks with matching cards.
        result = []
        if size_of_defender_hand > len(cards_to_match):
            num_of_players = len(self.players)
            # every player except the defender matches attack cards
            for i in range(num_of_players):
                player = self.players[(i + 1) % num_of_players]
                # get player's attack
                players_attack_cards = player.attack(size_of_defender_hand)
                # for each card: match if possible
                for card in cards_to_match:
                    new_card = player.match_cards(card, players_attack_cards,
size_of_defender_hand)
                    if not(new_card is None):
                        result.append(new_card)
                    if size_of_defender_hand == len(result):
                        return result
        return result


    # returns last player in the game (loser)
```

```
    def loser(self):
        return self.players[0]


    def check_player_status(self):
        for player in self.players:
            if player.is_empty:
                index = self.players.index(player)
                del self.players[index]


    def draw_cards(self):
        for player in self.players:
            while self.deck.size() != 0 and len(player.hand) < 6:
                player.add_to_hand(self.deck.pop_from_deck())


    # returns a list of all attackers
    def attackers(self):
        attacker_list = []
        for i in range(1, len(self.players)):
            attacker_list.append(self.players[i])
        return attacker_list


    # Takes the cards currently played. Allows each player to place additional
    # cards onto the table.  Returns updated list
    def add_to_table(self, attacker_list, cards_on_table):
        placed_cards = []
        for attacker in attacker_list:
            placed_cards.extend(attacker.match_cards(cards_on_table))
        return placed_cards


class Deck:

    def __init__(self):
        self.deck = []

    def shuffle(self):
        if self.size() == 0:
            random.shuffle(suits)
            for suit in suits:
                for j in range(6,15):
                    self.deck.append(Card(suit,j))
        random.shuffle(self.deck)
        return self.deck
```

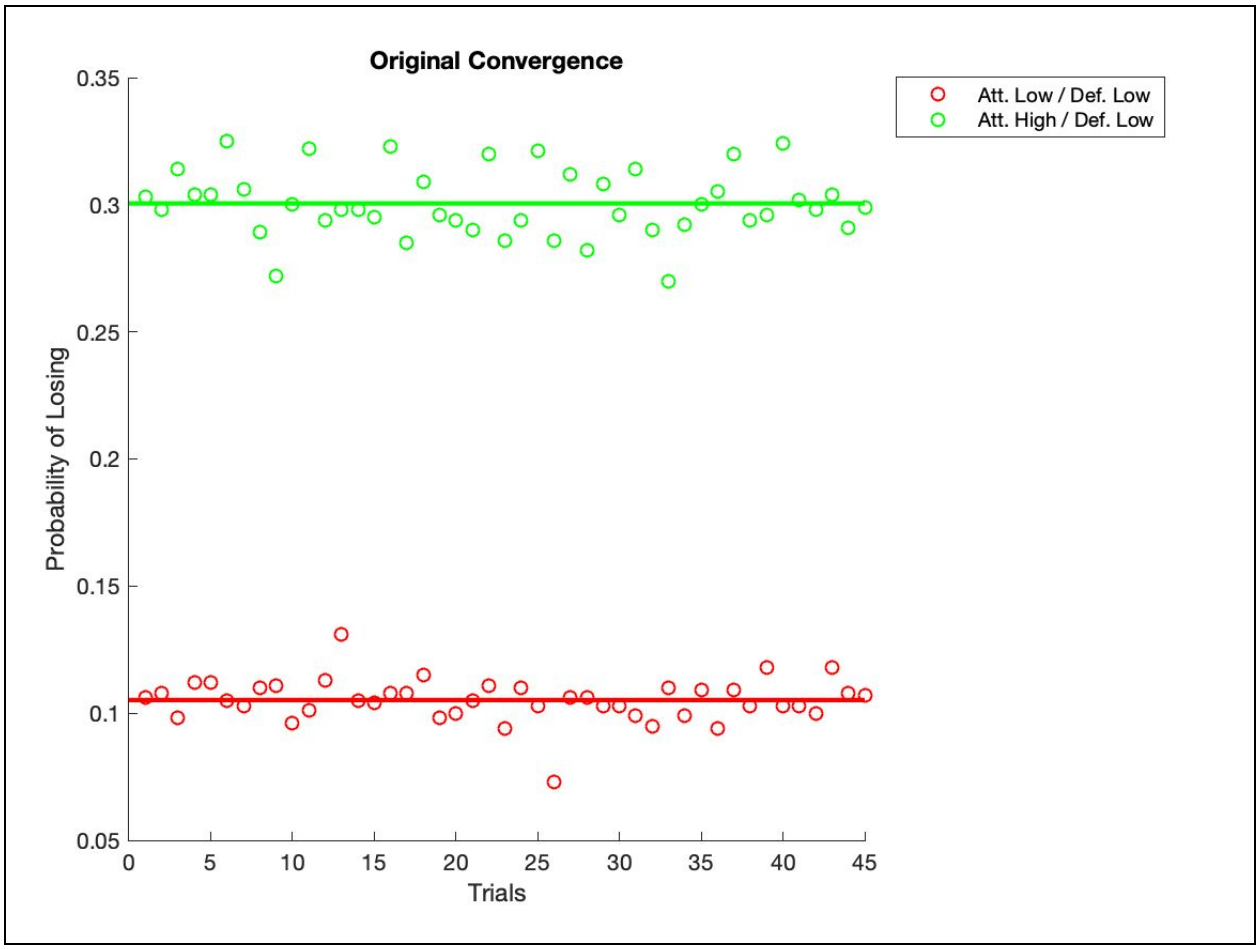
```
    def size(self):
        return len(self.deck)


    def get_trump_suit(self):
        return self.deck[0].suit


    def pop_from_deck(self):
        return self.deck.pop()


    def __repr__(self):
        return "({})".format(self.deck)


# finds lowest card in a given hand and returns it
def attack_with_low(hand):
    return get_minimum_cards(hand)


# returns the set of maximum cards for a given hand (strategy)
def attack_with_high(hand):
    return get_maximum_cards(hand)


# prioritizes trump cards low-high before normal cards (strategy)
def defend_with_trump(offense, hand):
    defend = set()
    trump = trump_list(hand)
    attack_trump_list = trump_list(offense)
    trump.sort()
    hand.sort()
    offense.sort()
    for card in offense:
        for play in hand:
            if ((card.suit == trump_suit and card.rank < play.rank) or ((card.suit
!= trump_suit)) and play.suit == trump_suit):
                defend.add(play)
    defend = list(defend)
    return defend


# returns all trump cards of a given hand
def trump_list(hand):
```

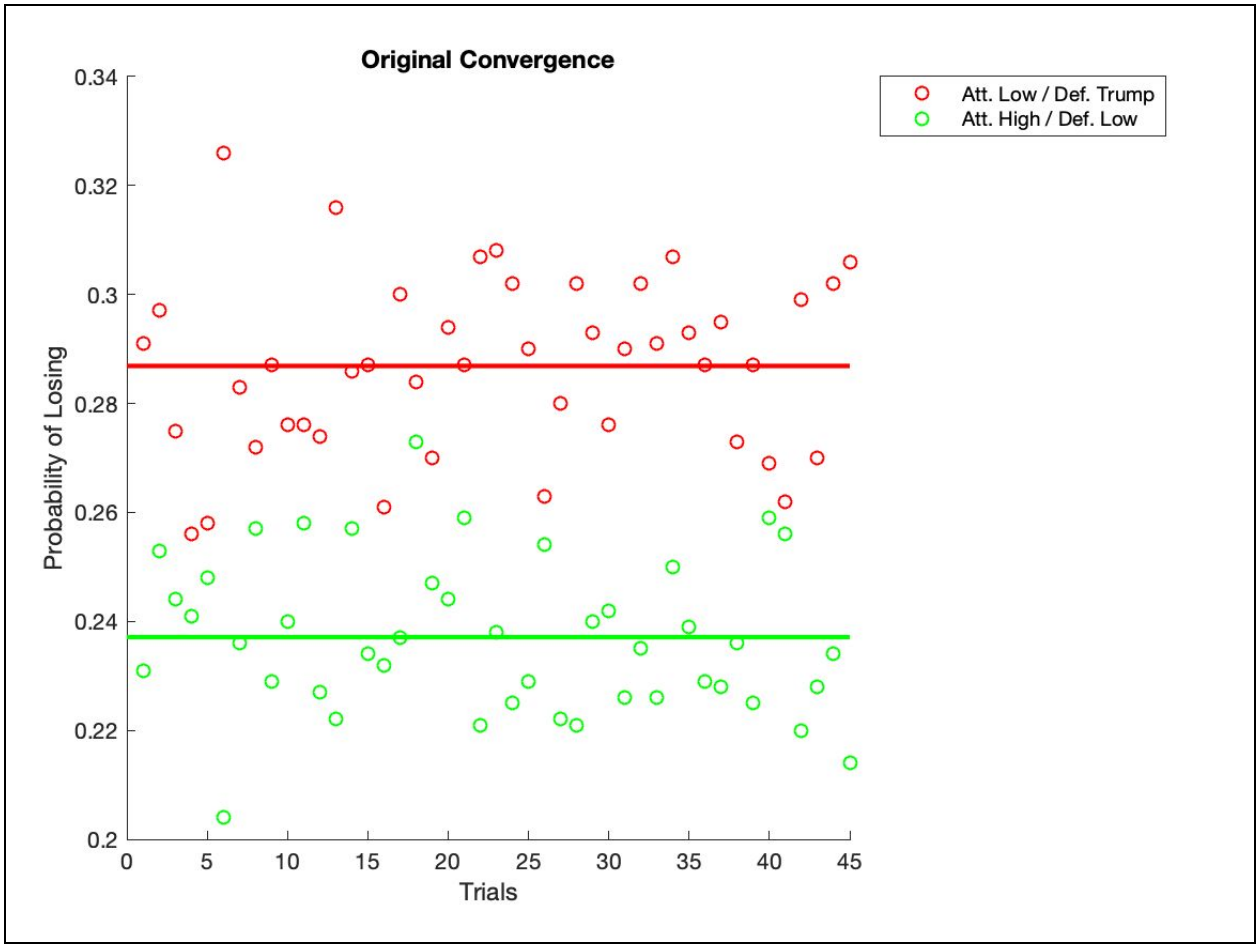
```
    trump = []
    for card in hand:
        if card.suit == trump_suit:
            trump.append(card)
    return trump


#return a list of cards.  Removes cards from hand (strategy)
def defend_with_low(offense, hand):
    defend = set()
    hand.sort()
    offense.sort()
    for card in offense:
        for play in hand:
            if card.suit == play.suit and card.rank < play.rank:
                defend.add(play)
                break
    defend = list(defend)
    return defend


# returns lowest value cards in a hand.  Works for an attack strategy
def get_minimum_cards(hand):
    # find min
    min_card = min(hand)
    return min_card

# returns maximum value cards in a hand.  Works for an attack strategy
def get_maximum_cards(hand):
    max_card = max(hand)
    return max_card

# records the person who loses in each game and exports it into a csv file
class Stats:

    def __init__(self, x):
        self.tracker = [0] * x

    def record(self, num):
        value = self.tracker[num - 1]
        self.tracker[num - 1] = value + 1

    def print_results(self, n):
        new_list = [float(x) / n for x in self.tracker]
        with open('results.csv', 'wb') as writefile:
            wr = csv.writer(writefile, quoting=csv.QUOTE_ALL)
            wr.writerow(new_list)
```

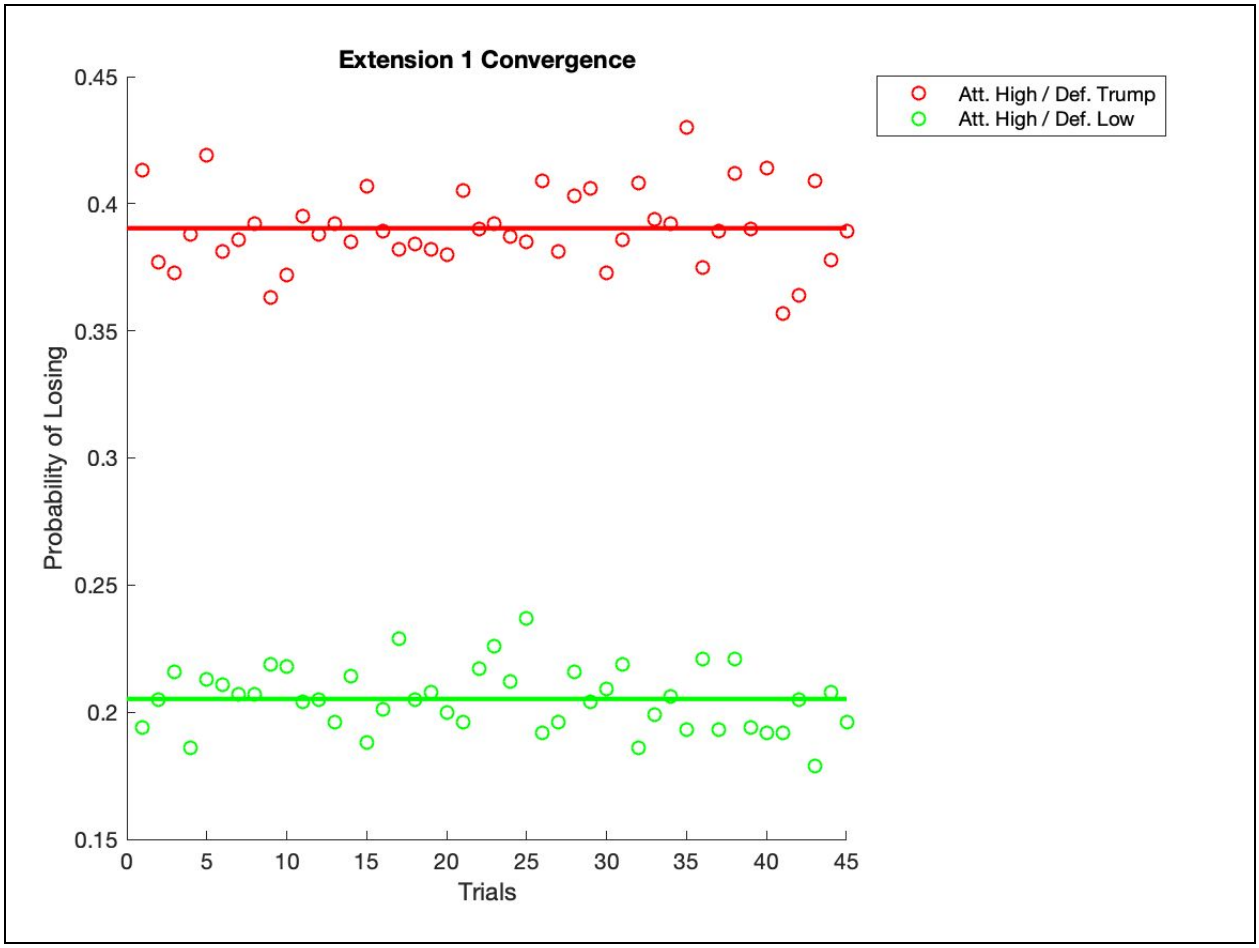
```
        csvFile.close()

    def record_overall(self, index, n, other):
        new_list = [float(x) / n for x in other]
        self.tracker[index] = new_list


def main(argv):

    times = 45
    overall = Stats(times)


    for index in range(times):

        number_of_players = 4
        stats = Stats(number_of_players)

        n = 1000
        for i in range(n):

            global trump_suit
            deck = Deck()
            deck.shuffle()
            trump_suit = deck.get_trump_suit()
            game_one = Game(deck)
            game_one.add_players()
            game_one.deal()

            while not(game_one.game_over()):
                game_one.play_round()

            loser = game_one.loser()

            stats.record(loser.num)

        overall.record_overall(index, n, stats.tracker)

    with open('defender_add_to_hand.csv', 'wb') as writefile:
        wr = csv.writer(writefile, quoting=csv.QUOTE_ALL)
        wr.writerow(['AL DL(add to hand)', 'AH DL', 'AH DL', 'AH DL'])
        for row in overall.tracker:
            wr.writerow(row)
        wr.writerow(['Average:'])
        wr.writerow(['=AVERAGE(A2:A46) * 100', '=AVERAGE(B2:B46) *
100','=AVERAGE(C2:C46) * 100','=AVERAGE(D2:D46) * 100'])
```

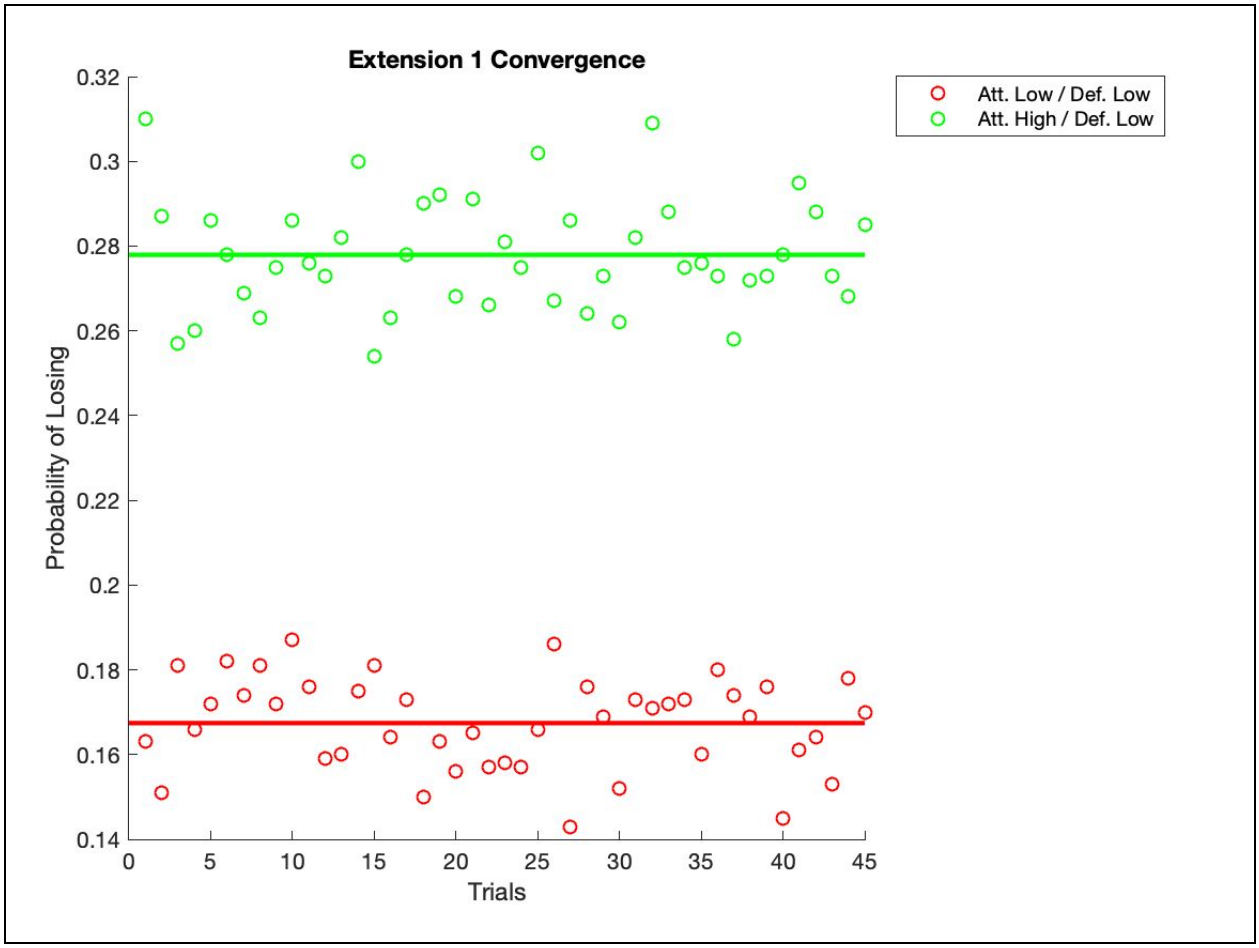
```
if __name__ == '__main__':
    main(sys.argv[1:])
```

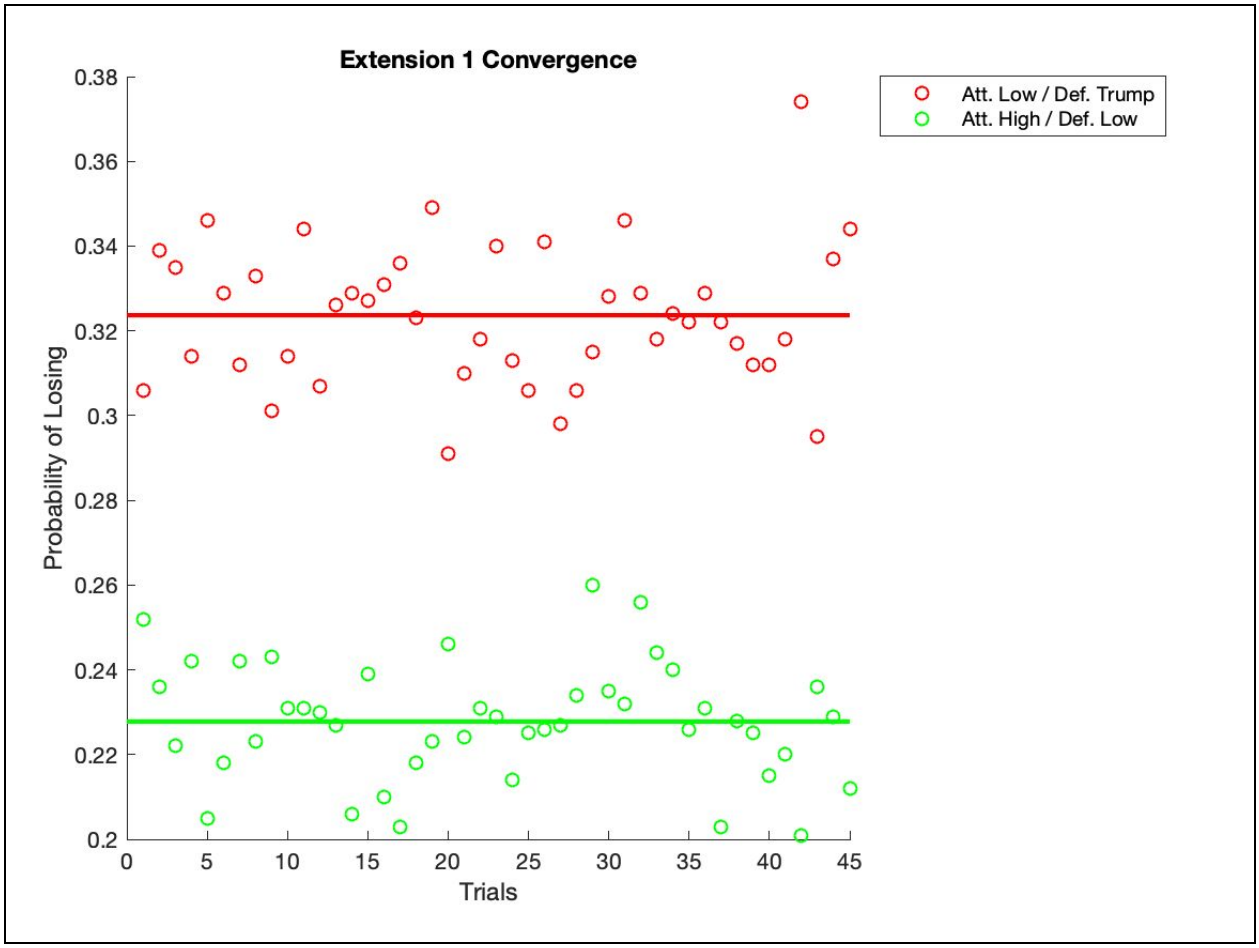
Plots

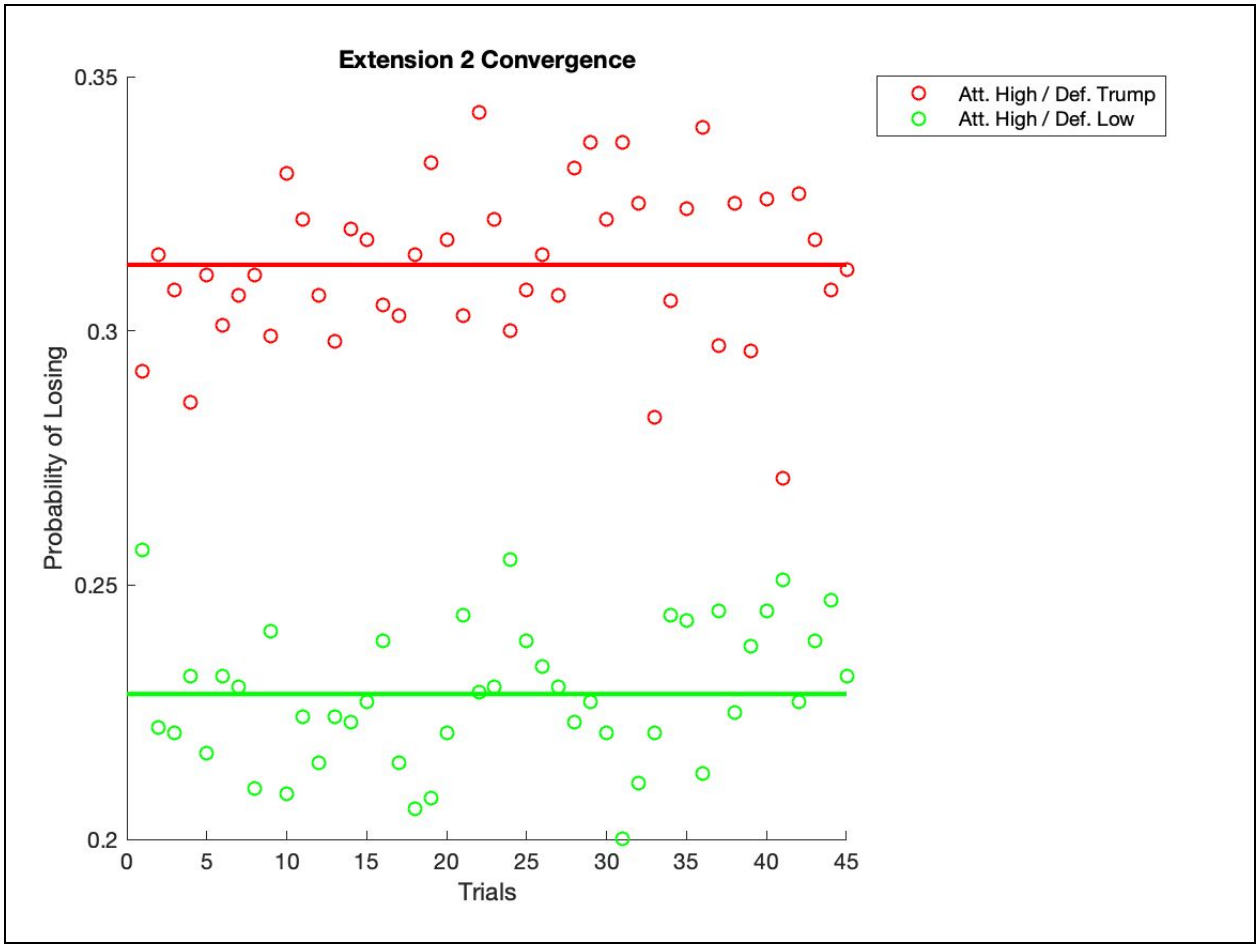


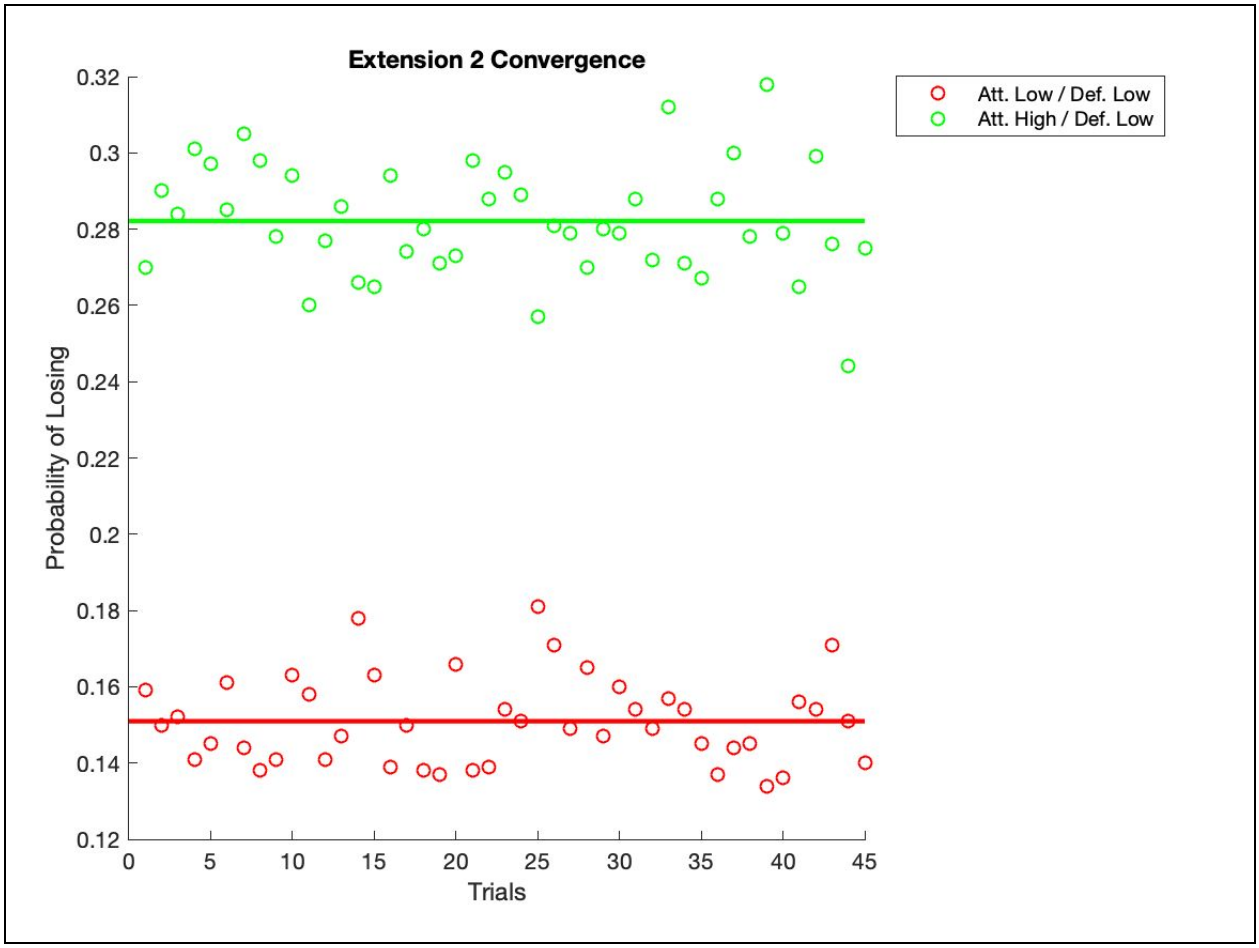

Original Convergence
Att. Low / Def. Trump
Att. High / Def. Low
Probability of Losing
Trials
0.34
0.32
0.3
0.28
0.26
0.24
0.22
0.2
0
5
10
15
20
25
30
35
40
45

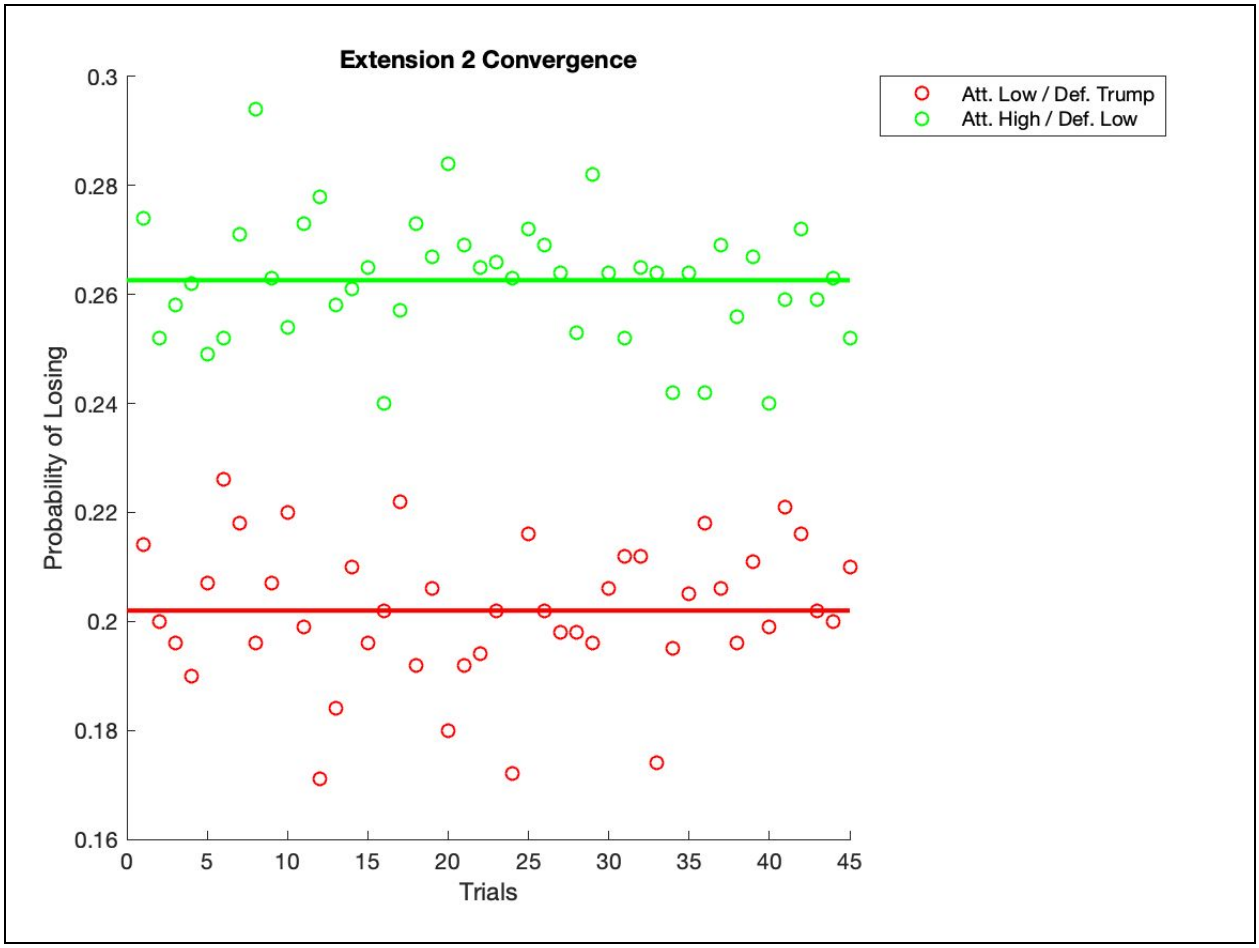

Extension 1 Convergence
Att. High / Def. Trump
Att. High / Def. Low
Probability of Losing
Trials
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0
5
10
15
20
25
30
35
40
45


Extension 1 Convergence
Att. Low / Def. Low
Att. High / Def. Low
Probability of Losing
Trials
0.32
0.3
0.28
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0
5
10
15
20
25
30
35
40
45


Extension 1 Convergence
Att. Low / Def. Trump
Att. High / Def. Low
Probability of Losing
Trials
0.38
0.36
0.34
0.32
0.3
0.28
0.26
0.24
0.22
0.2
0
5
10
15
20
25
30
35
40
45


Extension 2 Convergence
Att. High / Def. Trump
Att. High / Def. Low
Probability of Losing
0.35
0.3
0.25
0.2
0
5
10
15
20
25
30
35
40
45
Trials


Extension 2 Convergence
Att. Low / Def. Low
Att. High / Def. Low
Probability of Losing
0.32
0.3
0.28
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0.12
0
5
10
15
20
25
30
35
40
45
Trials


Extension 2 Convergence
Att. Low / Def. Trump
Att. High / Def. Low
Probability of Losing
Trials
0.3
0.28
0.26
0.24
0.22
0.2
0.18
0.16
0
5
10
15
20
25
30
35
40
45