Background

Movies and television have long glorified the idea of “beating Vegas.” While many strategies to take on this daunting task have been presented, the most popular, and possibly proven, technique is that of counting cards. While film has lead the population to believe that this is an overly complicated process that only a genius could pull off, it is actually quite simple. The most basic of all models is the “Hi-Lo” system. Invented by mathematician, Harry Dubner, in the 1960s, Hi-Lo is a simple plusminus strategy that assigns values to every card in the deck (2-6 = +1, 7-9 = 0 and 10-A = -1). The initial “count” starts at zero, then with every card dealt, that value is added to what is known as the running count. The running count is then dived by the number of decks remaining to give the “true count.” While counting cards does not make you win more hands, it does however tell you when you are likely to get a winning hand in the form of a positive count, allowing one to place a higher wager. With this strategy at our disposal, the questions become:

“How effective is this system?” and “What type of betting pattern is most successful?”

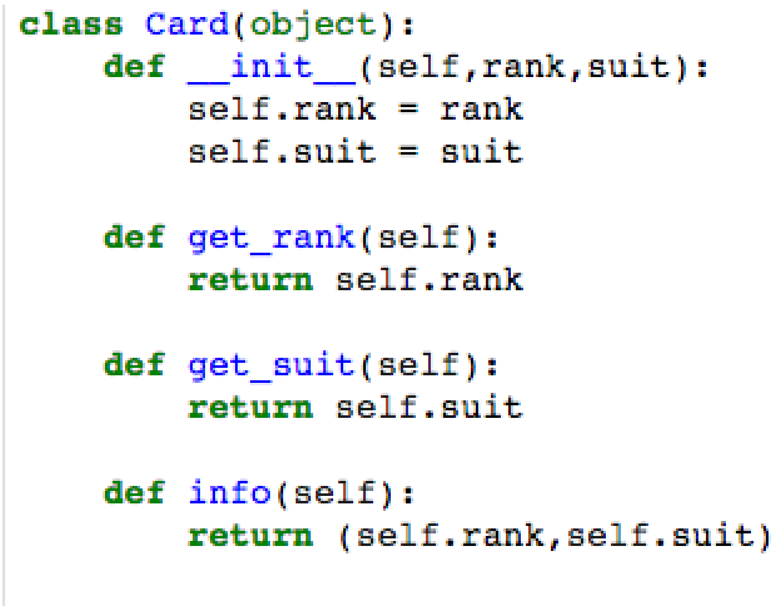
Methods

- The bulk of the simulation was created through object-oriented programming. Other, more familiar methods were also included in the creation of the algorithm used.

- Several parameters were tested

    - Initial small/large bets
        - Initial starting amount was constant
    - Number of hands played

- Classes and Objects
    - An example of one of the objects used, Card, can be seen below:



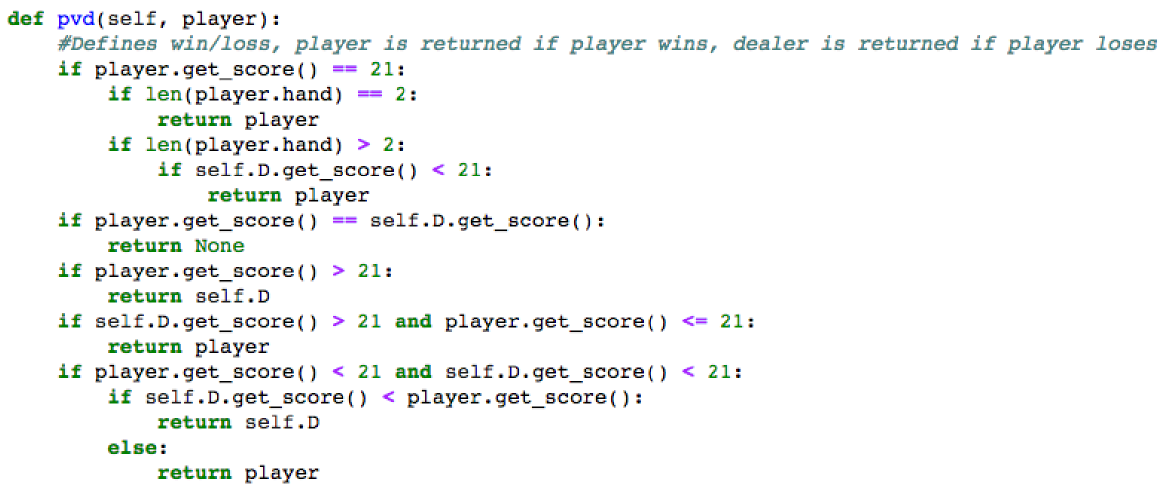
```python
class Card(object):
    def __init__(self,rank,suit):
        self.rank = rank
        self.suit = suit

    def get_rank(self):
        return self.rank

    def get_suit(self):
        return self.suit

    def info(self):
        return (self.rank,self.suit)
```

- Additionally, we defined an objects: Deck (a collection of all our cards), Agents—which contained the Player and Dealer subclasses—governed how both the dealer and players actually played, and Table, which is where our simulation ran. We populated our table with a certain number of players and shuffled decks, as well as a dealer, and then let our players and dealer play according to the rules we previously defined. One important aspect of this was determining who won and who lost. The code for this is shown below:

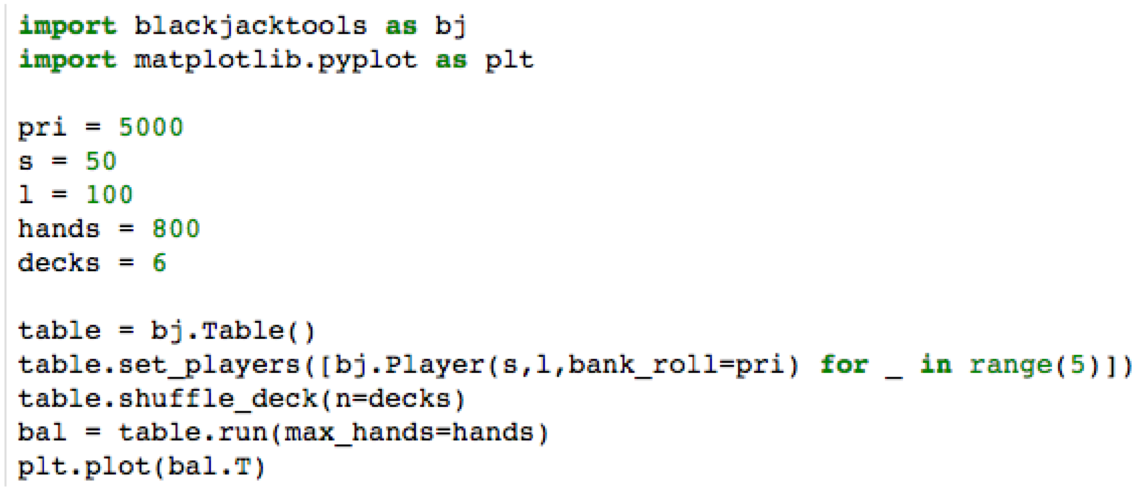


```python
def pvd(self, player):
    #Defines win/loss, player is returned if player wins, dealer is returned if player loses
    if player.get_score() == 21:
        if len(player.hand) == 2:
            return player
        if len(player.hand) > 2:
            if self.D.get_score() < 21:
                return player
    if player.get_score() == self.D.get_score():
        return None
    if player.get_score() > 21:
        return self.D
    if self.D.get_score() > 21 and player.get_score() <= 21:
        return player
    if player.get_score() < 21 and self.D.get_score() < 21:
        if self.D.get_score() < player.get_score():
            return self.D
        else:
            return player
```

- Most important function was the run function, which was defined within Table.
  - Allowed simulation to run, given the large and small bets,

the maximum number of hands, and the number of players

- Simulation returned end balances for each player
- We were able to create a driver that allowed us to quickly change the parameters to ensure that the simulation ran correctly. The driver is shown below.



```python
import blackjacktools as bj
import matplotlib.pyplot as plt

pri = 5000
s = 50
l = 100
hands = 800
decks = 6

table = bj.Table()
table.set_players([bj.Player(s,l,bank_roll=pri) for _ in range(5)])
table.shuffle_deck(n=decks)
bal = table.run(max_hands=hands)
plt.plot(bal.T)
```

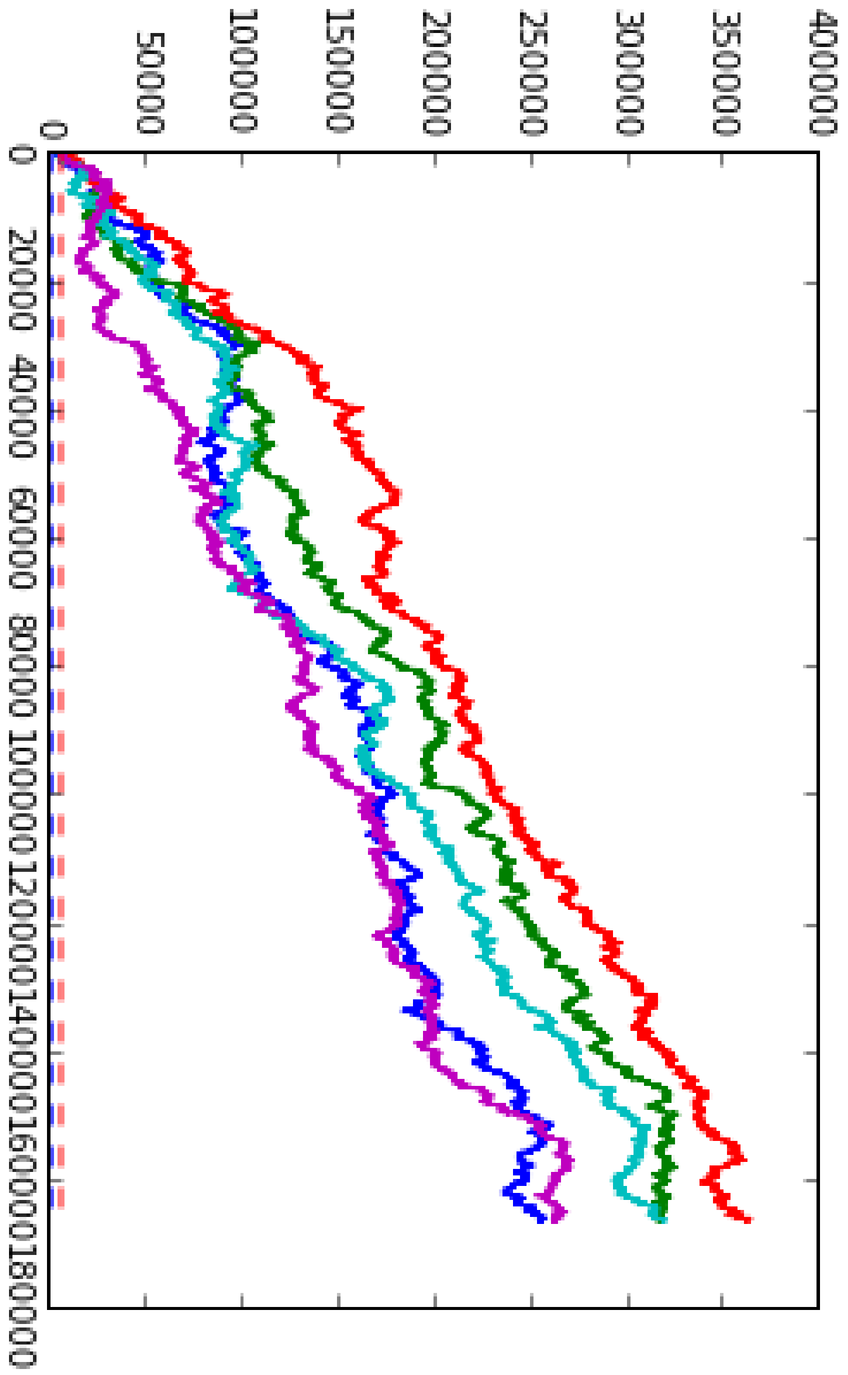
- We also needed to perform a large number of simulations with every possible permutation of a lowest minimum bet, which we set at $50, as well as a maximum bet, which we set at $500. In order to quickly analyze the data, we used a parameter sweep, and had each simulation return a plot of the player balances.

```
import blackjacktools as bj
import matplotlib.pyplot as plt

M = 150
Hmax = 500
Lmin = 50
for H in range(M,Hmax,5):
    for L in range(Lmin,M,5):
        table = bj.Table()
        table.shuffle_deck(n=6)
        table.set_players([bj.Player(L,H,bank_roll = 5000) for _ in range(5)])
        bal = table.run(max_hands = 800)
        plt.title("{} {}".format(L,H))
        plt.plot(bal.T)
        plt.ylim((0,25000))
        plt.savefig("img/{}_{}.pdf".format(L,H))
        plt.clf()
```

Results

Graph 1 (Good Scheme)


0
50000
100000
150000
200000
250000
300000
350000
400000
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

Axis Titles:

Number of Hands

Player Bankroll (USD)

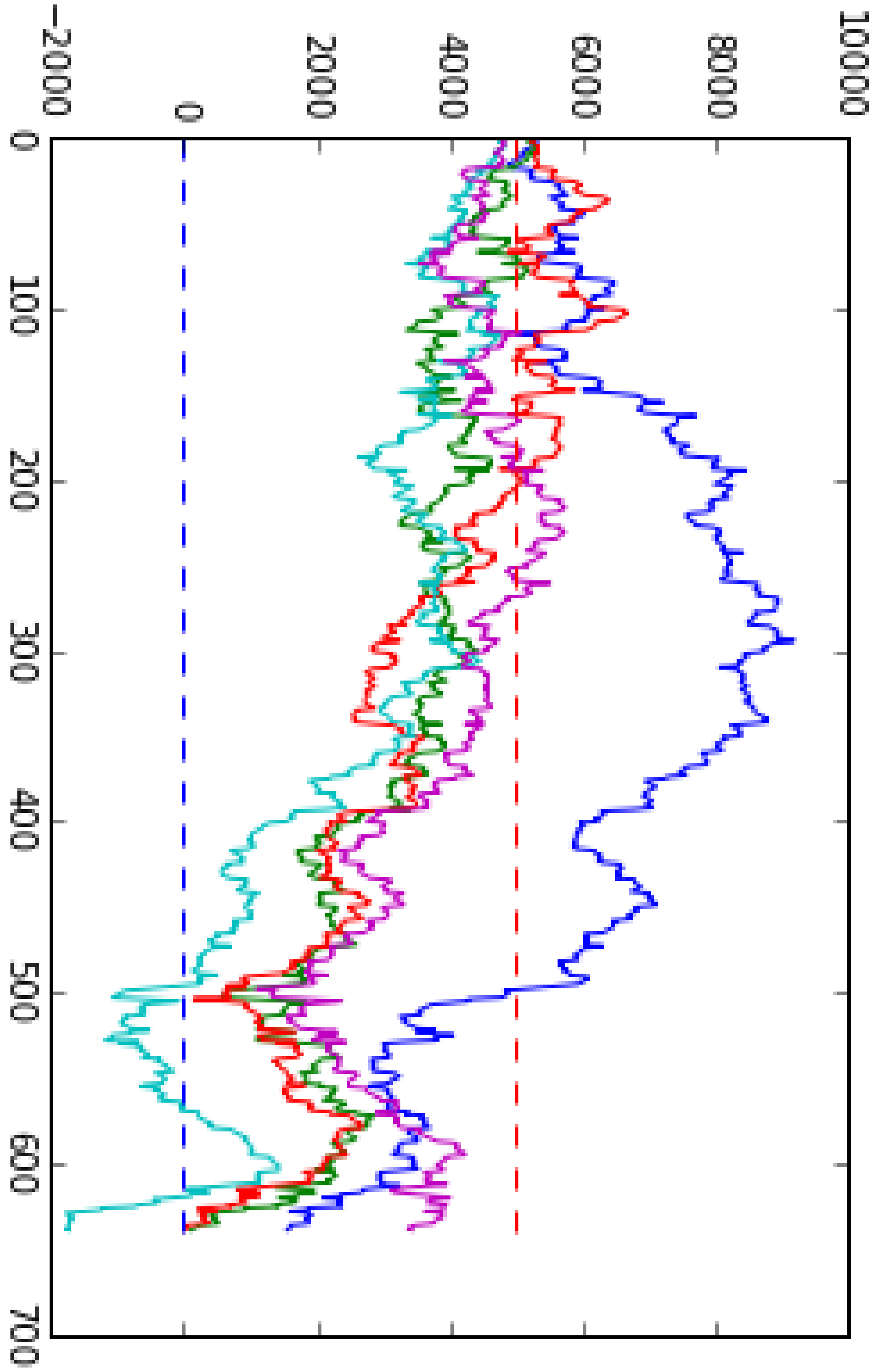
Graph Title:

1 Year Simulation of a Good Betting Scheme

Graph Caption:

In this simulation, each player had a small bet of $145, and a large bet of $165. Each player starts off with $5000, and after a year’s worth of hands, every player has made between $250,000 and $350,000.

Graph 2 (Bad Scheme)


−2000
0
2000
4000
6000
8000
10000
0
100
200
300
400
500
600
700

Axis Titles:

Number of Hands

Player Bankroll (USD)

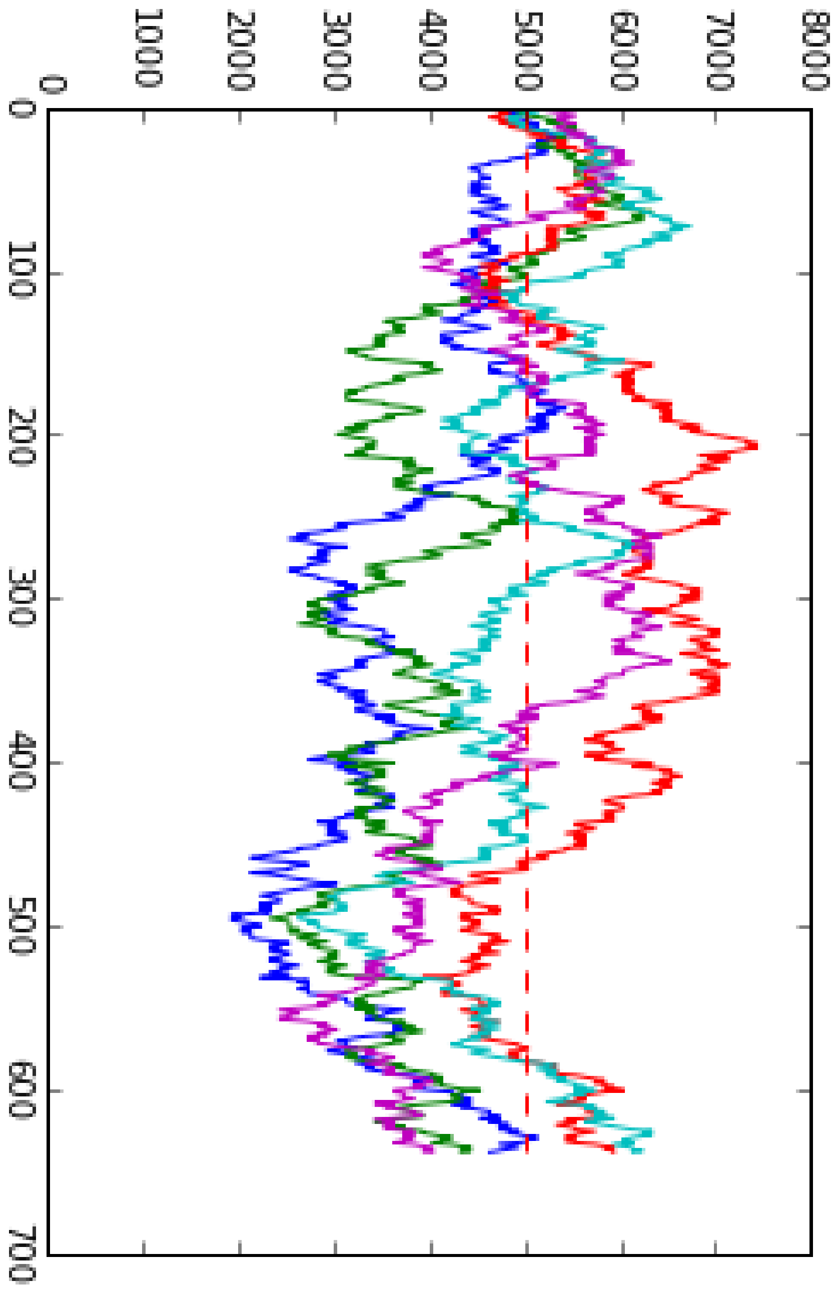
Graph Title:

1 Day Simulation of a Poor Betting Scheme

Graph Caption:

In this simulation, each player had a small bet of $50, and a large bet of $500. Each player starts off with $5000, and after a day’s worth of hands, every player has lost money, with three players losing all of their money.

Graph 3 (~Average Scheme):


0
1000
2000
3000
4000
5000
6000
7000
8000
0
100
200
300
400
500
600
700

Axis Titles:

Number of Hands

Player Bankroll (USD)

Graph Title:

1 Day Simulation of an Average Betting Scheme

Graph Caption:

In this simulation, each player had a small bet of $100, and a large bet of $300. Each player starts off with $5000, and after a day's worth of hands, three of the players have lost money, while the other two made money. While we were unable to implement a proper control, we know that most casino games are set up such that the house always wins. This betting scheme gave similar results, so we have used it as a control.

Summary

Using object-oriented programming, our team was able to successfully simulate thousands of hands of blackjack with a variety of betting patterns. By running all of these simulations, we were able to find a few ideal betting patterns, as well as some not-so-ideal ones. While we were unable to properly analyze all of patterns, a quick examination of our results revealed that conservative betting schema, where there was a very small difference between the largest and smallest bet were far more lucrative and consistent than aggressive betting strategies, where the difference between the large and small bets was quite large. We were able to obtain our results, despite the fact that our simulation involves a very basic way of playing blackjack. Within our algorithm, a player hits until their score is seventeen or greater, at which point they stay. In reality, playing blackjack is quite more complicated. Players typically determine their playing and betting habits off of the dealer’s hand as well as their own. Additionally, there are a few additional strategies, such as splitting a player’s hand, or doubling down, that we were unable to integrate into our simulation due to time constraints.

In the future, we would like to add all of these features, as well as all of the other strategies that are listed on a typical blackjack strategy chart. We would also like to implement some sort of control player that plays without counting cards. Only then will we be able to properly demonstrate that counting cards is a better strategy than not counting cards. However, given our results, it seems apparent that counting cards is indeed a viable strategy. By refining our simulation such that each player plays in a more realistic and complicated manner, we expect that one could make even more money over the course of a single year.

Acknowledgements

We wish to thank Ian Kenney for his teaching of object oriented programming and help with various problems along the way.