Reinforcement learning simple experiment -- Blackjack

- *Objective*: Have your card sum be greater than the dealers without exceeding 21.
- All face cards are counted as 10, and the ace can count either as 1 or as 11.
- *States*:
  - current sum (12-21)
  - dealer's showing card (ace, 2-10)
  - do I have a usable ace?
- *Reward*: +1 for winning, 0 for a draw, -1 for losing (not discount)
- *Actions*: stick (stopping receiving cards), hit (receive another card)
- *Policy*: stick if my sum is 20 or 21, else hit

If the player holds an ace that he could count as 11 without going bust, then the ace is said to be *usable*.

- Dealer's fixed strategy

STICK if $>=$ 17

HIT if $<$ 17

- Outcome:

if $>$ 21 $=>$ LOSE

CLOSEST to 21 $=>$ WIN

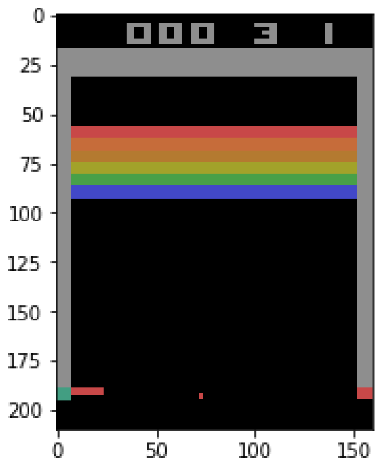
EQUALLY CLOSE $=>$ DRAW

Reference

1. Render gym in python notebook https://stackoverflow.com/questions/40195740/how-to-run-openai-gym-render-over-a-server/40579087 (https://stackoverflow.com/questions/40195740/how-to-run-openai-gym-render-over-a-server/40579087)
2. blackjack https://applenob.github.io/black_jack.html (https://applenob.github.io/black_jack.html)
3. Monte Carlo Policy Evaluation http://www.incompleteideas.net/book/ebook/node51.html (http://www.incompleteideas.net/book/ebook/node51.html)
4. Reinforcement learning https://github.com/dennybritz/reinforcement-learning (https://github.com/dennybritz/reinforcement-learning)
5. On-Policy Monte Carlo Control http://www.incompleteideas.net/book/ebook/node54.html (http://www.incompleteideas.net/book/ebook/node54.html)
6. Off-Policy Monte Carlo Control http://www.incompleteideas.net/book/ebook/node56.html (http://www.incompleteideas.net/book/ebook/node56.html)

Test the environment and define plotting functions

In [1]:
```
import gym
from IPython import display
import numpy as np
import pandas as pd
from collections import namedtuple
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict
%matplotlib inline

EpisodeStats = namedtuple('Stats', ['episode_lengths', 'episode_rewards'])


def plot_value_function(V, title = "Value Function"):
    '''
    Plots the value function as a surface plot.
    '''
    min_x = 11 # min(k[0] for k in V.keys())
    max_x = max(k[0] for k in V.keys())
    min_y = min(k[1] for k in V.keys())
    max_y = max(k[1] for k in V.keys())

    x_range = np.arange(min_x, max_x + 1)
    y_range = np.arange(min_y, max_y + 1)
    X, Y = np.meshgrid(x_range, y_range)

    # Find value for all (x, y) coordinates
    Z_noace = np.apply_along_axis(lambda _: V[(_[0], _[1], False)], 2, np.dstack([X, Y]))
    Z_ace = np.apply_along_axis(lambda _: V[(_[0], _[1], True)], 2, np.dstack([X, Y]))

    def plot_surface(X, Y, Z, title):
        fig = plt.figure(figsize = (20, 10))
        ax = fig.add_subplot(111, projection = '3d')
        surf = ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1,
                               cmap = matplotlib.cm.coolwarm, vmin = -1.0, vmax = 1.0)
        ax.set_xlabel('Player Sum')
        ax.set_ylabel('Dealer Showing')
        ax.set_zlabel('Value')
        ax.set_title(title)
        ax.view_init(ax.elev, -120)
        fig.colorbar(surf)
        plt.show()

    plot_surface(X, Y, Z_noace, "{} (No Usable Ace)".format(title))
    plot_surface(X, Y, Z_ace, "{} (Usable Ace)".format(title))

env = gym.make('Breakout-v0') # insert your favorite environment
env.reset()
img = plt.imshow(env.render(mode = 'rgb_array'))
for _ in range(100):
    img.set_data(env.render(mode = 'rgb_array'))
    display.display(plt.gcf())
    display.clear_output(wait = True)
    action = env.action_space.sample()
    env.step(action)
```


0
25
50
75
100
125
150
175
200
0
50
100
150

Blackjack environment

```python
import gym
from gym import spaces
from gym.utils import seeding

def cmp(a, b):
    return float(a > b) - float(a < b)

# 1 = Ace, 2-10 = Number cards, Jack/Queen/King = 10
deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]


def draw_card(np_random):
    return int(np_random.choice(deck))


def draw_hand(np_random):
    return [draw_card(np_random), draw_card(np_random)]


def usable_ace(hand):  # Does this hand have a usable ace?
    return 1 in hand and sum(hand) + 10 <= 21


def sum_hand(hand):  # Return current hand total
    if usable_ace(hand):
        return sum(hand) + 10
    return sum(hand)


def is_bust(hand):  # Is this hand a bust?
    return sum_hand(hand) > 21


def score(hand):  # What is the score of this hand (0 if bust)
    return 0 if is_bust(hand) else sum_hand(hand)


def is_natural(hand):  # Is this hand a natural blackjack?
    return sorted(hand) == [1, 10]


class BlackjackEnv(gym.Env):
    """Simple blackjack environment
    Blackjack is a card game where the goal is to obtain cards that sum to as
    near as possible to 21 without going over.  They're playing against a fixed
    dealer.
    Face cards (Jack, Queen, King) have point value 10.
    Aces can either count as 11 or 1, and it's called 'usable' at 11.
    This game is placed with an infinite deck (or with replacement).
    The game starts with each (player and dealer) having one face up and one
    face down card.
    The player can request additional cards (hit=1) until they decide to stop
    (stick=0) or exceed 21 (bust).
    After the player sticks, the dealer reveals their facedown card, and draws
    until their sum is 17 or greater.  If the dealer goes bust the player wins.
    If neither player nor dealer busts, the outcome (win, lose, draw) is
    decided by whose sum is closer to 21.  The reward for winning is +1,
    drawing is 0, and losing is -1.
    The observation of a 3-tuple of: the players current sum,
    the dealer's one showing card (1-10 where 1 is ace),
    and whether or not the player holds a usable ace (0 or 1).
    This environment corresponds to the version of the blackjack problem
    described in Example 5.1 in Reinforcement Learning: An Introduction
    by Sutton and Barto (1998).
    http://incompleteideas.net/sutton/book/the-book.html
    """
    def __init__(self, natural=False):
        self.action_space = spaces.Discrete(2)
        self.observation_space = spaces.Tuple((
            spaces.Discrete(32),
            spaces.Discrete(11),
            spaces.Discrete(2)))
        self.seed()

        # Flag to payout 1.5 on a "natural" blackjack win, like casino rules
        # Ref: http://www.bicyclecards.com/how-to-play/blackjack/
        self.natural = natural
        # Start the first game
        self.reset()

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def step(self, action):
        assert self.action_space.contains(action)
        if action:  # hit: add a card to players hand and return
            self.player.append(draw_card(self.np_random))
            if is_bust(self.player):
                done = True
                reward = -1
            else:
                done = False
                reward = 0
        else:  # stick: play out the dealers hand, and score
            done = True
            while sum_hand(self.dealer) < 17:
                self.dealer.append(draw_card(self.np_random))
            reward = cmp(score(self.player), score(self.dealer))
            if self.natural and is_natural(self.player) and reward == 1:
                reward = 1.5
        return self._get_obs(), reward, done, {}

    def _get_obs(self):
        return (sum_hand(self.player), self.dealer[0], usable_ace(self.player))

    def reset(self):
        self.dealer = draw_hand(self.np_random)
        self.player = draw_hand(self.np_random)
        return self._get_obs()
```

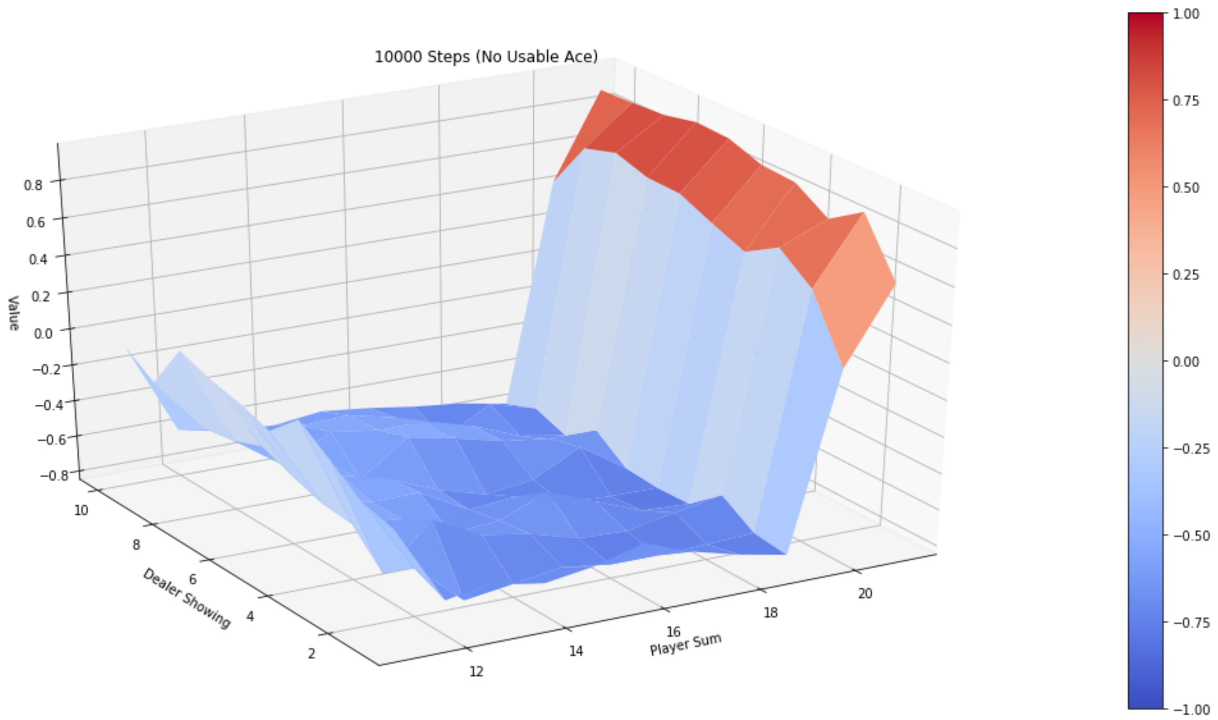
Monte Carlo Policy Evaluation

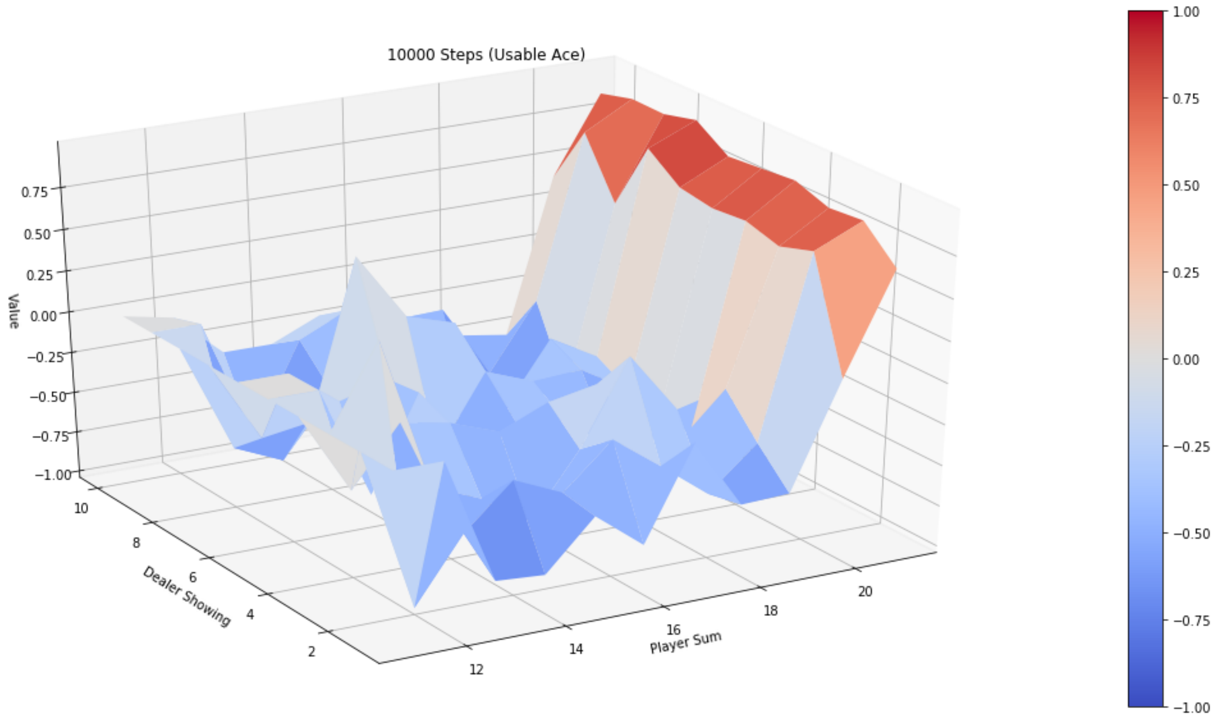
We begin by considering Monte Carlo methods for learning the state-value function for a given polcy. Recall that the value of a state is the expected return -- expected cumulative future discounted reward -- starting from the state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more return are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

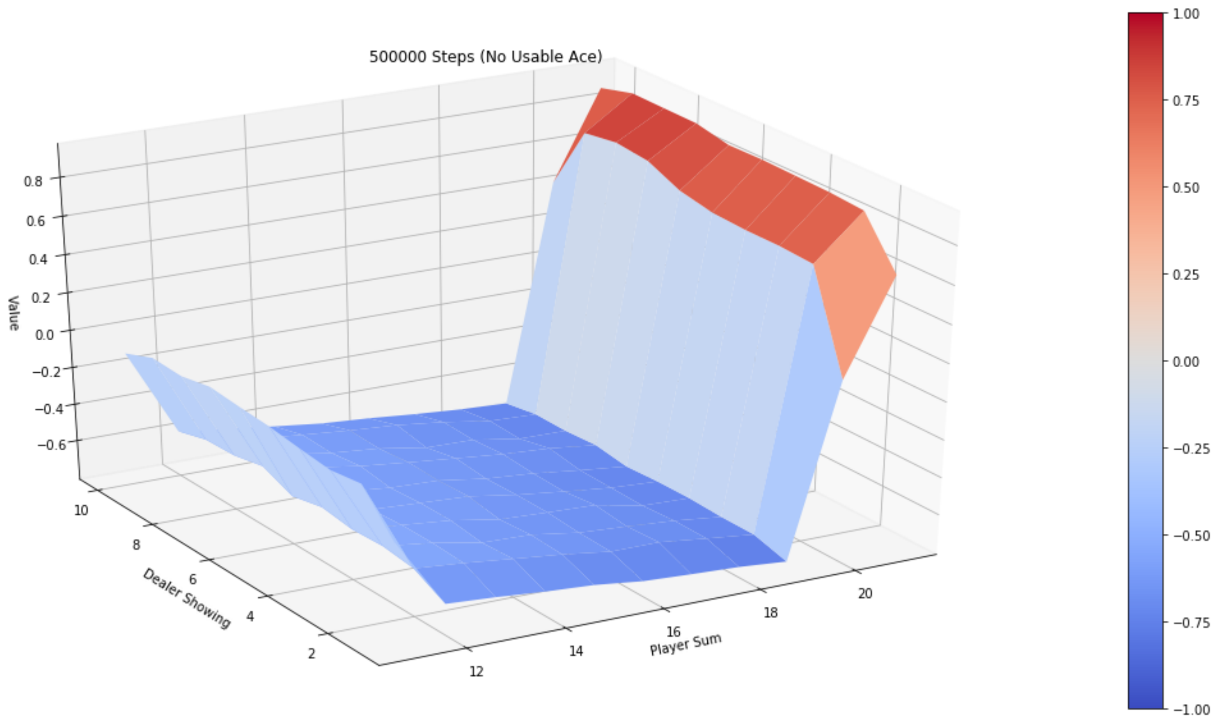
In [2]:
```
def mc_prediction(policy, env, num_episodes, discount_factor = 1.0):
    '''
    Monte Carlo prediction algorithm.
    Calculates the value function for a given policy using sampling.

    Args:
        policy: A function that maps an observation to action probabilities.
        env: OpenAI gym environment.
        num_episodes: Nubmer of episodes to sample.
        discount_factor: Lambda discount factor.

    Returns:
        A dictionary that maps from state -> value.
        The state is a tuple and the value is a float.
    '''
    # Keeps track of sum and count of returns for each state
    # to calculate an average. We could use an array to save all
    # returns but that's memory inefficient.
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The final value function
    V = defaultdict(float)


    for i_episode in range(num_episodes):
        observation = env.reset()

        episodes = []
        for i in range(100):
            action = policy(observation)
            next_observation, reward, done, _ = env.step(action)
            episodes.append((observation, action, reward))
            if done:
                break
            observation = next_observation

        # obtain unique observation set
        observations = set([x[0] for x in episodes])
        for i, observation in enumerate(observations):
            # first occurence of the observation
            idx = episodes.index([episode for episode in episodes if episode[0] == observa
tion][0])

            Q = sum([episode[2] * discount_factor ** i for episode in episodes[idx:]])

            returns_sum[observation] += Q
            returns_count[observation] += 1.0

            V[observation] = returns_sum[observation] / returns_count[observation]

    return V
```

In [3]:
```
env = gym.make('Blackjack-v0')

# A policy that sticks if the player score is >= 20 and hits otherwise.
def sample_policy(observation):
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1
```

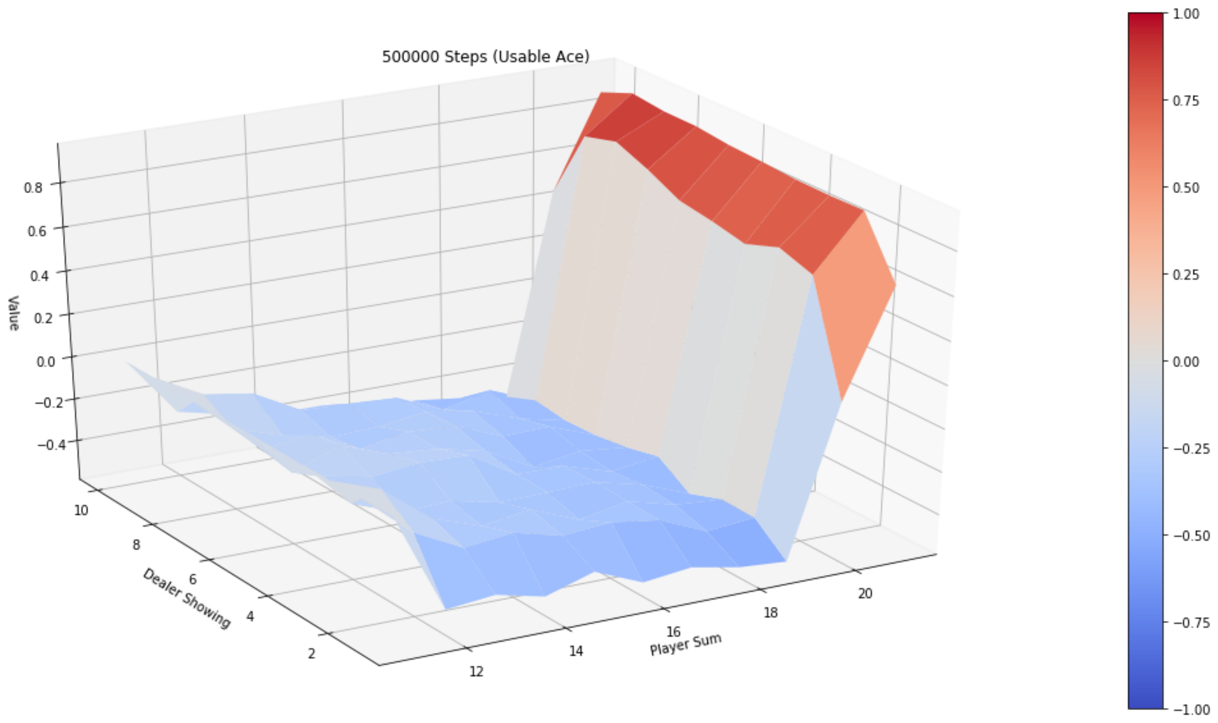
In [4]:
```
V_10k = mc_prediction(sample_policy, env, num_episodes = 10000)
plot_value_function(V_10k, title = '10000 Steps')

V_500k = mc_prediction(sample_policy, env, num_episodes = 500000)
plot_value_function(V_500k, title = '500000 Steps')
```


10000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


10000 Steps (Usable Ace)
Value
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



500000 Steps (No Usable Ace)
Value
Dealer Showing
Player Sum

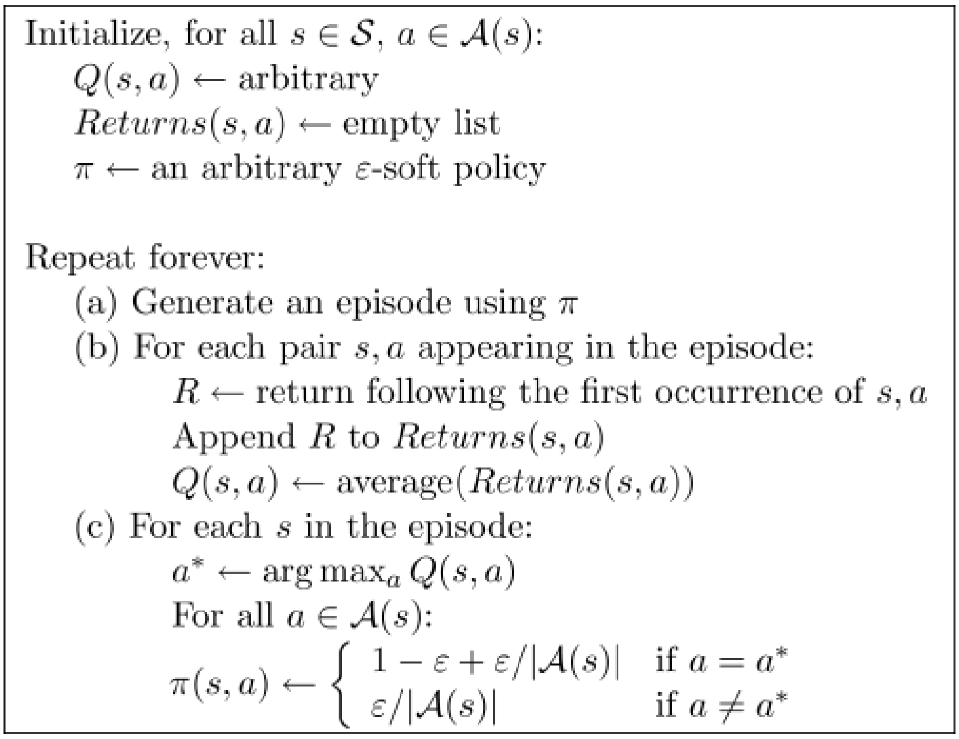


500000 Steps (Usable Ace)
Value
Dealer Showing
Player Sum

As we can see there's only a small winning rate no matter what the number the dealer's showing is and what the player's sum is for this strategy.

On-Policy Monte Carlo Control

In on-policy control methods the policy is generally *soft*, meaning that $\pi(s, a) > 0$ for all $s \in S$ and all $s \in A(s)$. There are many possible variations on on-policy methods. One possibility is to gradually shift the policy toward a deterministic optimal policy. The on-policy method we present in this section use ϵ-greedy policies, meaning that most of the time they choose an action that has maximum estimated action value, but with propability ϵ they instead select an action at random. That is, all nongreedy actions are gievn the minimal propability of selection, $\frac{\epsilon}{|A(s)|}$, and the remaining bulk of the probability, $1 - \epsilon + \frac{\epsilon}{|A(s)|}$, is given to the greedy action.

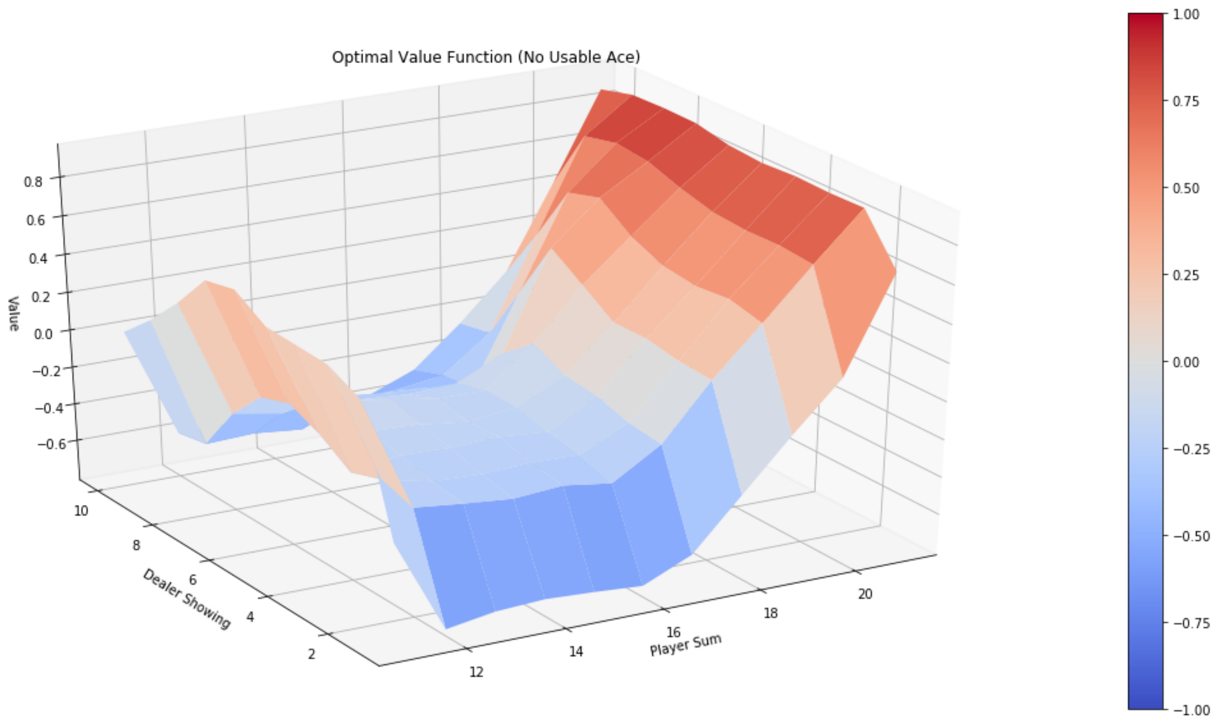


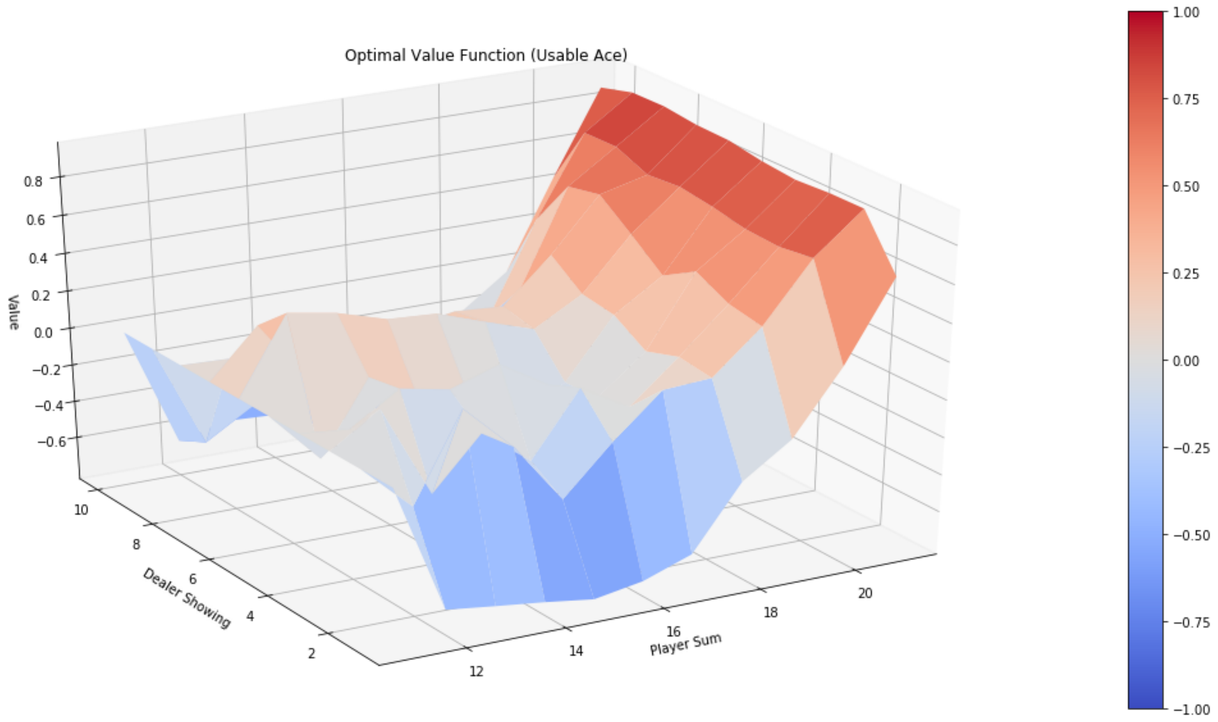
```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each pair $s, a$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s, a$
            Append $R$ to $Returns(s,a)$
            $Q(s,a) \leftarrow$ average$(Returns(s,a))$
    (c) For each $s$ in the episode:
            $a^* \leftarrow \arg\max_a Q(s,a)$
            For all $a \in \mathcal{A}(s)$:
            $\pi(s,a) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
```

In [8]:
```
def make_epsilon_greedy_policy(Q, epsilon, nA):
    '''
    Creates an epsilon-greedy policy based on a given Q-function and epsilon.

    Args:
        Q: A dictionary that maps from state -> action-values.
            Each value is a numpy array of length nA (see below)
        epsilon: The probability to select a random action . float between 0 and 1.
        nA: Number of actions in the environment.

    Returns:
        A function that takes the observation as an argument and returns
        the probabilities for each action in the form of a numpy array of length nA.

    '''
    def policy_fn(observation):
        # Implement this!
        props = np.ones(nA, dtype = float) * epsilon / nA
        best_action = np.argmax(Q[observation])
        props[best_action] += 1. - epsilon
        return props

    return policy_fn
```

In [9]:
```python
def mc_control_epsilon_greedy(env, num_episodes, discount_factor = 1.0, epsilon = 0.1):
    '''
    Monte Carlo Control using Epsilon-Greedy policies.
    Finds an optimal epsilon-greedy policy.

    Args:
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        discount_factor: Gamma discount factor.
        epsilon: Chance the sample a random action. Float betwen 0 and 1.

    Returns:
        A tuple (Q, policy).
        Q is a dictionary mapping state -> action values.
        policy is a function that takes an observation as an argument and returns
        action probabilities
    '''

    # Keeps track of sum and count of returns for each state
    # to calculate an average. We could use an array to save all
    # returns (like in the book) but that's memory inefficient.
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The final action-value function.
    # A nested dictionary that maps state -> (action -> action-value).
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # The policy we're following
    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    # Implement this!
    for i_episode in range(num_episodes):
        observation = env.reset()

        episodes = []
        for i in range(100):
            props = policy(observation)
            action = np.random.choice(np.arange(len(props)), p = props)
            next_observation, reward, done, _ = env.step(action)
            episodes.append((observation, action, reward))

            if done:
                break
            observation = next_observation

        # find the unique observation
        pairs = set([(episode[0], episode[1]) for episode in episodes])
        for (observation, action) in pairs:
            pair = (observation, action)
            # find the first occurence of the observation
            idx = episodes.index([episode for episode in episodes if episode[0] == observa
tion and episode[1] == action][0])
            V = sum([episode[2] * discount_factor ** i for i, reward in enumerate(episodes
[idx:])])

            returns_sum[pair] += V
            returns_count[pair] += 1.

            Q[observation][action] = returns_sum[pair] / returns_count[pair]

    return Q, policy
```

```
In [10]: Q, policy = mc_control_epsilon_greedy(env, num_episodes = 500000, epsilon = 0.1)

         # For plotting: Create value function from action-value function
         # by picking the best action at each state
         V = defaultdict(float)
         for state, actions in Q.items():
             action_value = np.max(actions)
             V[state] = action_value
         plot_value_function(V, title = 'Optimal Value Function')
```

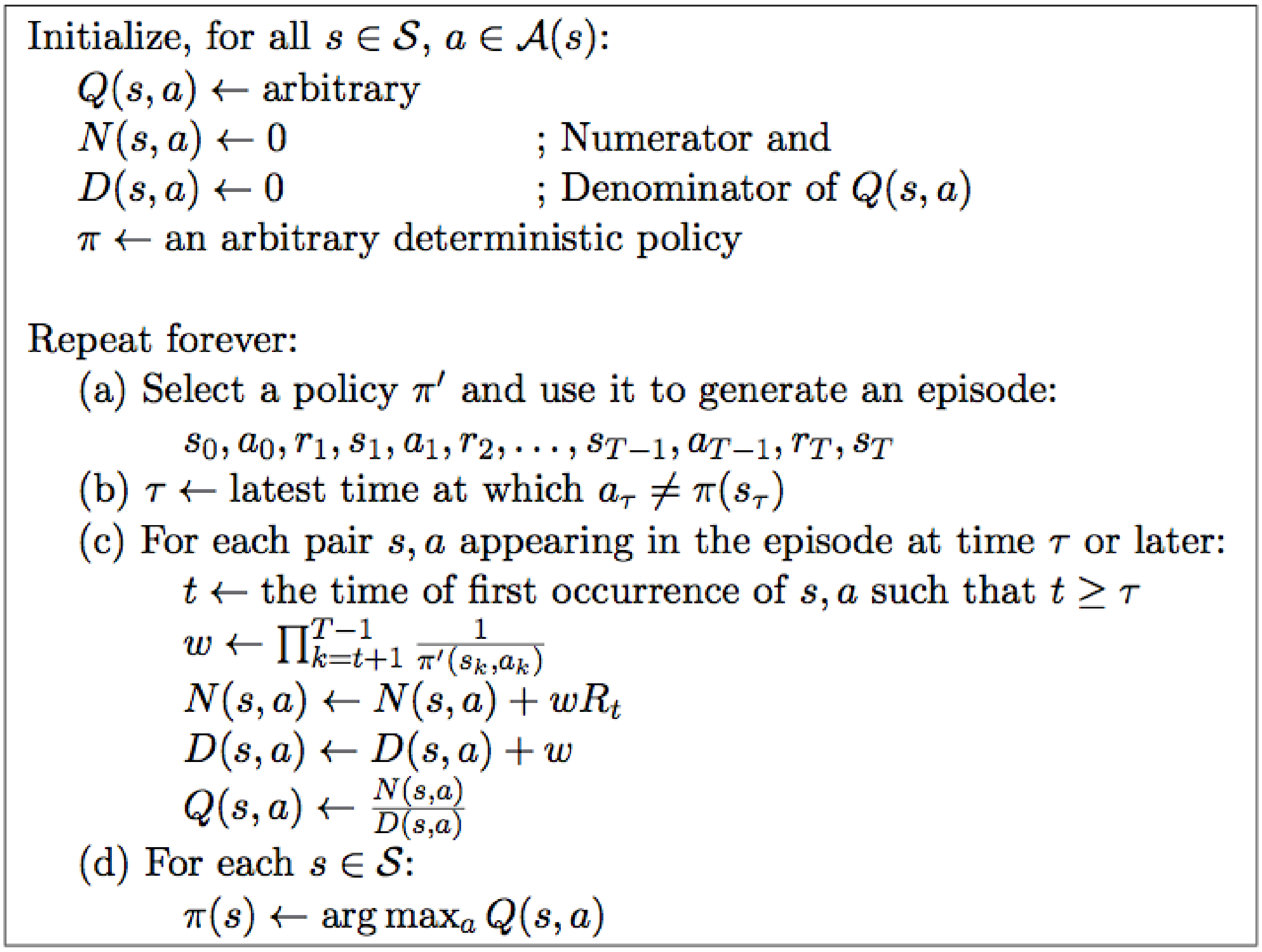




Comparing the rerults, there is a higher winner rate of ϵ-greedy policy than the previous policy.

Off-Policy Monte Carlo with weighted importance sampling

Recall that the distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy method, these functinos are separated. The policy used to generate behavior, called the *bahavior* policy, may in fact by unrelated to the policy that is evaluated and improved, called the *estimation* policy. An advantage of this separation is that the estimation policy may be determinisitic (e.g. greedy), while the behavior policy can continue to sample all possible actions.



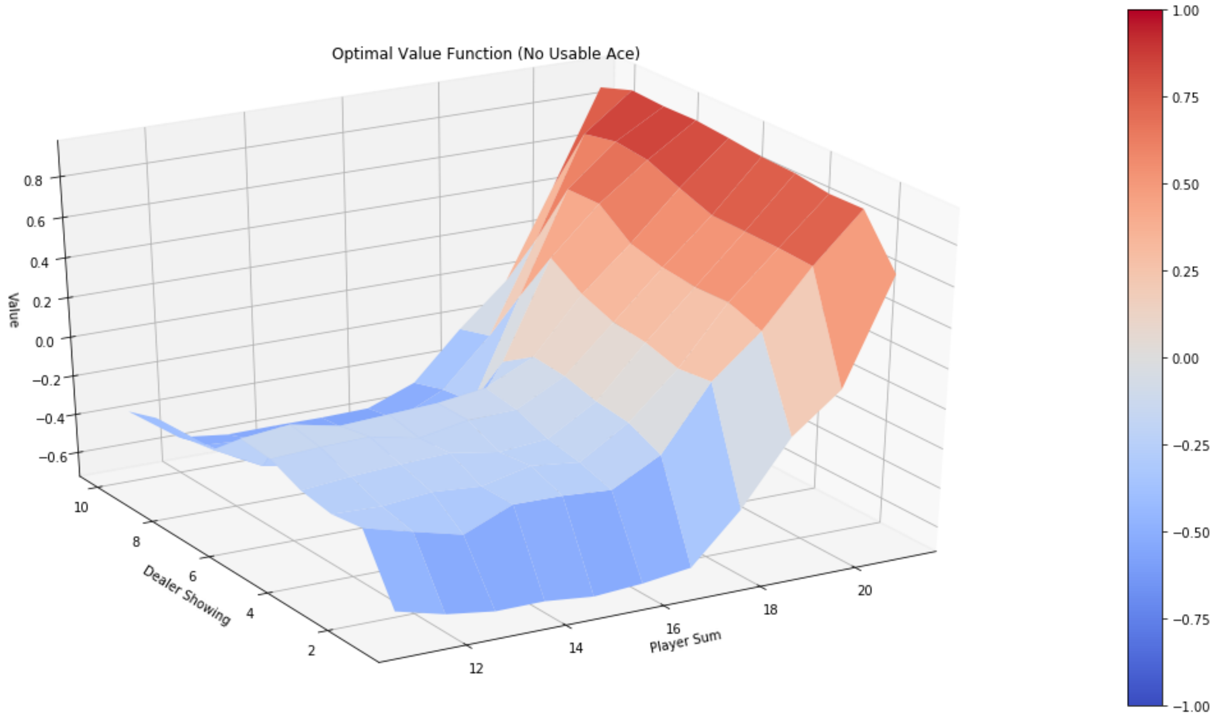
```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $N(s,a) \leftarrow 0$                ; Numerator and
    $D(s,a) \leftarrow 0$                ; Denominator of $Q(s,a)$
    $\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
    (a) Select a policy $\pi'$ and use it to generate an episode:
            $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
    (b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
    (c) For each pair $s, a$ appearing in the episode at time $\tau$ or later:
            $t \leftarrow$ the time of first occurrence of $s, a$ such that $t \geq \tau$
            $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k,a_k)}$
            $N(s,a) \leftarrow N(s,a) + wR_t$
            $D(s,a) \leftarrow D(s,a) + w$
            $Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
    (d) For each $s \in \mathcal{S}$:
            $\pi(s) \leftarrow \arg\max_a Q(s,a)$
```

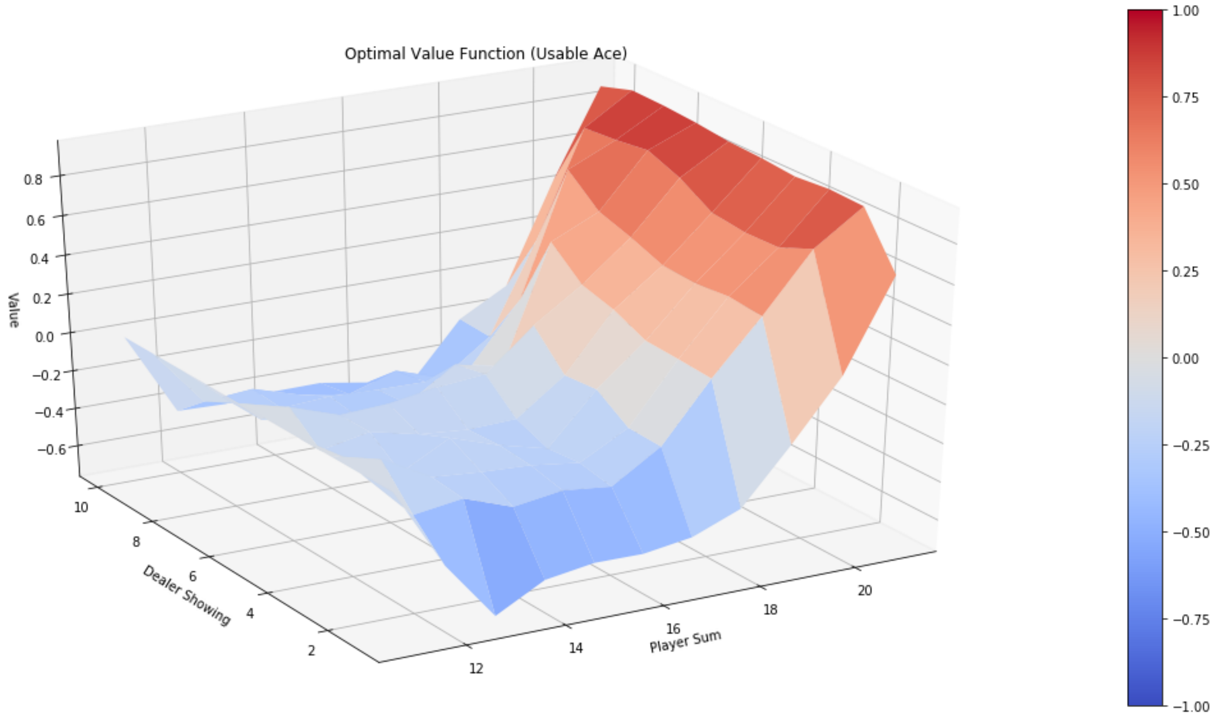
In [11]:
```
def create_random_policy(nA):
    '''
    Creates a random policy function.
    
    Args:
        nA: Number of actions in the environment.
        
    Returns:
        A function that takes an observation as input and returns a vector
        of action probabilities
    '''
    A = np.ones(nA, dtype = float) / nA
    def policy_fn(observation):
        return A
    return policy_fn
```

In [17]:
```
def create_greedy_policy(Q):
    '''
    Creates a greedy policy based on Q values.

    Args:
        Q: A dictionary that maps from state -> action values

    Returns:
        A function that takes an observation as input and returns a vector
        of action probabilities.
    '''

    def policy_fn(observation):
        # Implement this!
        props = Q[observation]
        best_action = np.argmax(props)
        A = np.zeros((len(props)), dtype = float)
        A[best_action] = 1.
        return A
    return policy_fn
```

In [26]:
```
def mc_control_importance_sampling(env, num_episodes, behavior_policy, discount_factor = 1.0):
    '''
    Monte Carlo Control Off-Policy Control using Weighted Importance Sampling.
    Finds an optimal greedy policy.

    Args:
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        behavior_policy: The behavior to follow while generating episodes.
            A function that given an observation returns a vector of probabilities for each action.
        discount_factor: Gamma discount factor.

    Returns:
        A tuple (Q, policy).
        Q is a dictionary mapping state -> action values.
        policy is a function that takes an observation as an argument and returns
        action probabilities. This is the optimal greedy policy.
    '''
    N = defaultdict(lambda: np.zeros(env.action_space.n))
    D = defaultdict(lambda: np.zeros(env.action_space.n))

    # The final action-value function.
    # A dictionary that maps state -> action values
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Our greedily policy we want to learn
    target_policy = create_greedy_policy(Q)

    # Implement this!
    for i_episode in range(num_episodes):
        observation = env.reset()

        episodes = []
        for i in range(100):
            props = behavior_policy(observation)
            action = np.random.choice(np.arange(len(props)), p = props)
            next_observation, reward, done, _ = env.step(action)
            episodes.append((observation, action, reward))
            if done:
                break
            observation = next_observation

        pairs = set([(episode[0], episode[1]) for episode in episodes])
        for (observation, action) in pairs:
            pair = (observation, action)

            # find the first occurence of this pair in episodes
            idx = episodes.index([episode for episode in episodes if episode[0] == observation and episode[1] == action][0])

            V = sum([episode[2] * discount_factor ** i for episode in episodes[idx:]])

            w = np.product([1. / behavior_policy(episode[0])[episode[1]] for episode in episodes[idx:]])

            N[observation][action] += w * V
            D[observation][action] += w

            Q[observation][action] = N[observation][action] / D[observation][action]

    return Q, target_policy
```

```
In [27]: random_policy = create_random_policy(env.action_space.n)
         Q, policy = mc_control_importance_sampling(env, num_episodes = 500000, behavior_policy = r
         andom_policy)

         # For plotting: Create value function from action-value function
         # by picking the best action at each state
         V = defaultdict(float)
         for state, action_values in Q.items():
             action_value = np.max(action_values)
             V[state] = action_value
         plot_value_function(V, title = 'Optimal Value Function')
```





Similar trend can be observed as the ϵ-greedy policy.