Easy 21 – Reinforcement Learning Assignment
John Goodacre – 21/03/16

Question 1

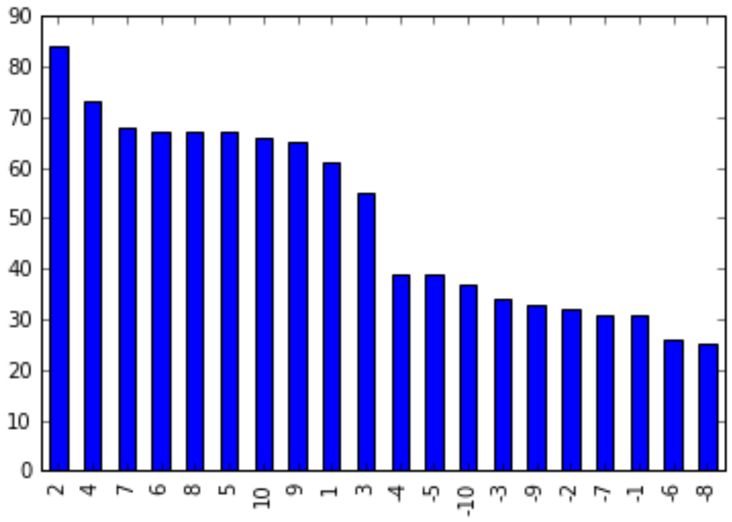
Write an environment that implements the game Easy21, create a draw function, a step function and provide output test files.

The assignment:
I wrote Easy21 in Python 2.7 in an IPython notebook (attached as easy21.ipynb). Also included are the files checkDraw.txt, checkStepDealer10Player15Action0.txt, checkStepDealer1Player10Action1.txt, checkStepDealer1Player1Action.txt, checkStepDealer1Player18Action0.txt, and finally for question 2 checkQ.txt. The code includes no non-standard python libraries and should run in one's browser.

Each cell in easy21 has been labelled to enable reference to the questions in the assignment. My apologies, I decided this would be a good opportunity to shift from Matlab and learn some Python along the way, hence the code is somewhat monolithic, ugly and not object oriented.

Question 1 simply asks us to implement the environment and provide test files. I have done this with the functions draw(), check_draw(), step(s,a), and check_step(s,a).

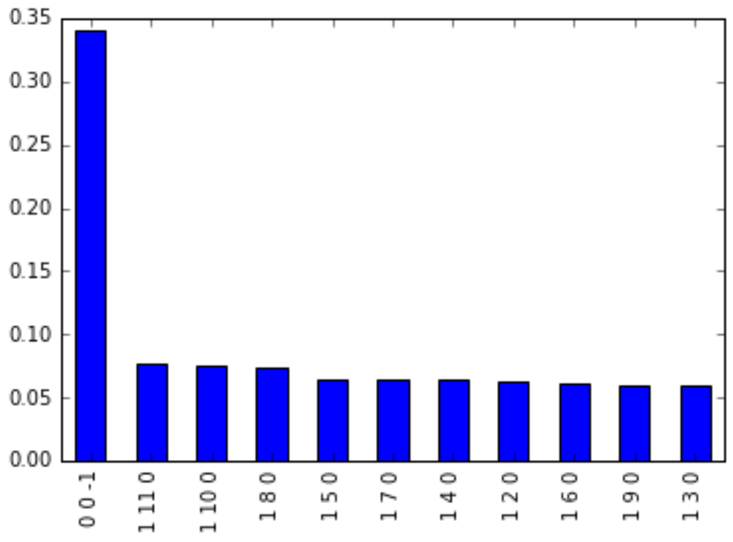
For example one run of check_draw (1000 hands) results in the following chart.



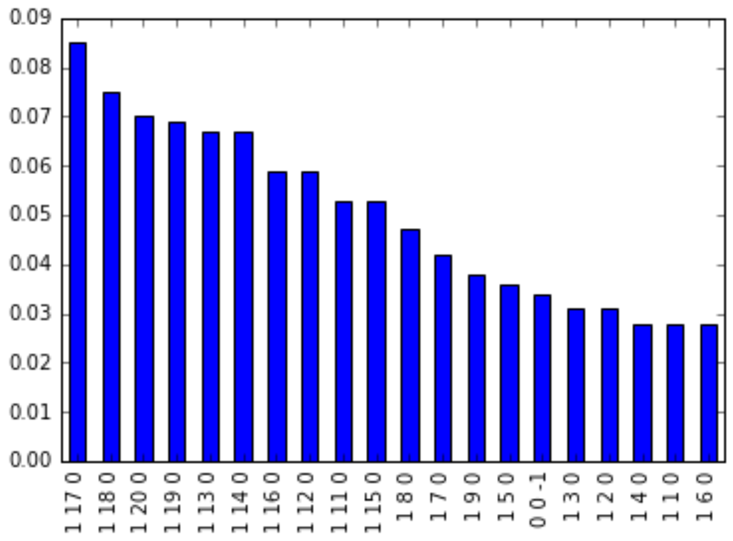
This shows a histogram of the number of times a card was drawn. Note that black (positive cards) are twice as likely as red.

Similarly charts for check_step() are provided. For example in the case check_step((1,1),HIT) we see the likelihood of the next state. The most likely in this case being (0,0,-1). The first two zeros mean we are now in a terminal state with the result being a loss. The first number of the tuple is the dealer up-card, the next the total for the player. A result of 0 is either non-terminal or a draw.

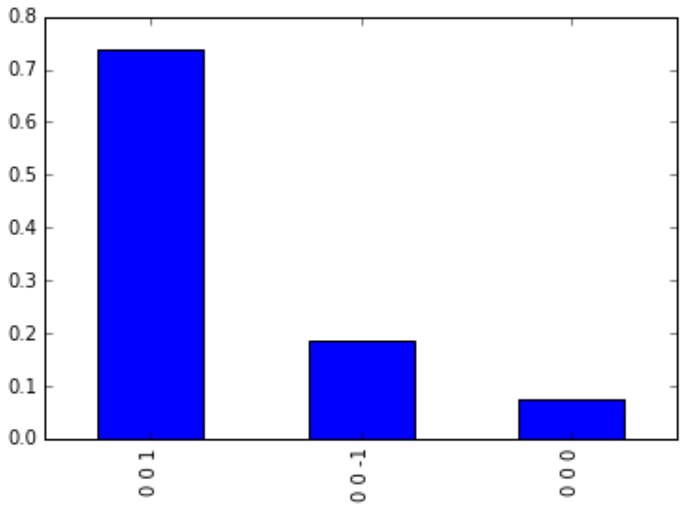
- check_step((1,1),HIT)



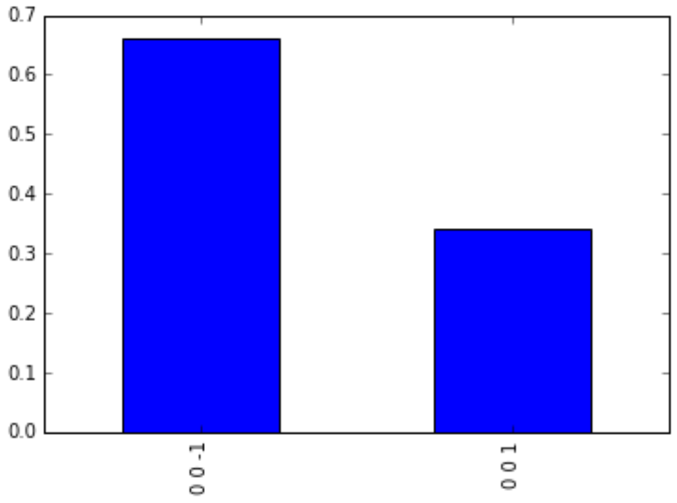
- check_step((1,10),HIT)



- check_step((1,18), STICK)



- check_step((10,15), STICK)



Question 2

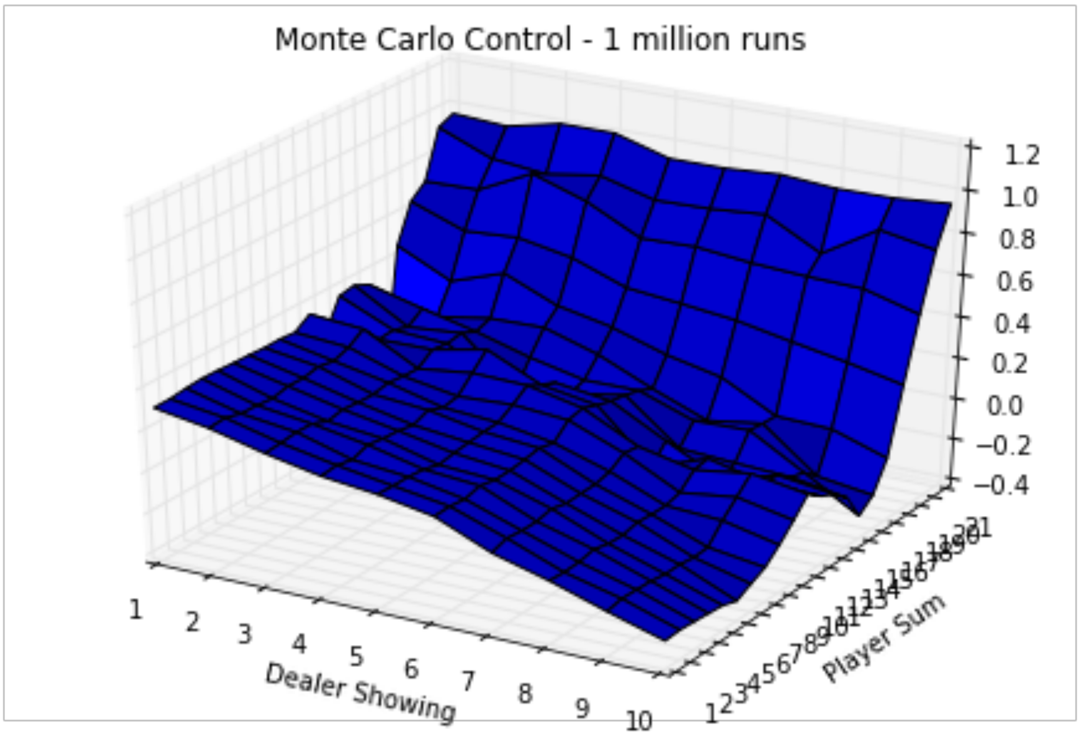
Apply Monte-Carlo control to Easy21. Initialise the value function to 0. Use a time varying step size of alpha being the number of times a state action tuple has been visited and use greedy exploration. Estimate and plot the optimal value function by doing a million runs.

The Monte-Carlo simulation is implemented using the following functions – starting-state(), start_policy(), action(), episode(), start_Q(), start_N(), with the real bulk of the work being done in update_QP(). The data-structures Q,P,N are dictionaries mapping a state-action tuple to a number. In the case of Q – the value function, in the case of N the number of times a state-action pair has been visited.

Update_QP() does both value and policy iteration, the intuition being that I generate episodes and then loop through each state-action tuple until I reach the end of an episode. The reward thus generated is propagated back through all state-actions visited this informing the algorithm of its best guesstimate of value and policy.

Epsilon-Greedy represents the ability of the algorithm to choose a random action with a low probability epsilon, rather than always choosing the greedy action, thus enabling exploration.

The update_V() and plot_value_functions() are helper functions to enable the display of the optimal value surface with monte() being where question 2 in the assignment is actually run from.



The figure above shows the optimal value surface for Monte-Carlo control and ties somewhat with intuition. For example a player total of 21 is very good across the board. Whereas a player total in the mid-teens is an exercise in damage limitation particularly against say a dealer up-card of a 10.

Clearly the surface becomes smoother as more episodes are performed. I use checkQ() to generate a text file giving the values of my state-action pairs after 1 million runs.

Question 3

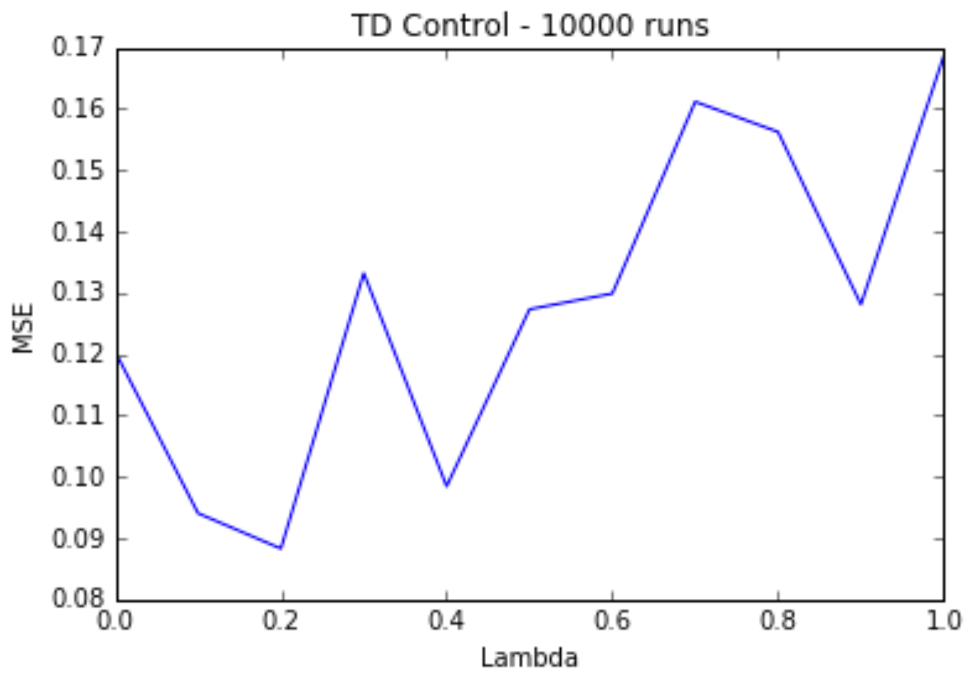
Implement Sarsa – Lambda for Easy21. Using the same step and exploration as before. Vary lambda and report the mean squared error after 10,000 runs. We are using the 1million run Monte-Carlo control to compare against (note in this case I reported mean-square error, not the sum of squared errors).

This was implemented in the rather ugly function update_QPTD() again my apologies for implementing this whilst learning Python. The intuition here is that again I update values and policies in one function, however rather than going to the end of each episode as in Monte-Carlo control instead I move to the next step and look at m.y reward there. And after each step update my whole state-space of values.

The eligibility trace ensures that relevant values are updated and the parameter lambda affects the fade of the eligibility trace.

In effect this is bootstrapping, because clearly most states are not terminal and so the reward is not known, however over multiple runs the information from the terminal states is still propagated back to enable learning of our Q surface.

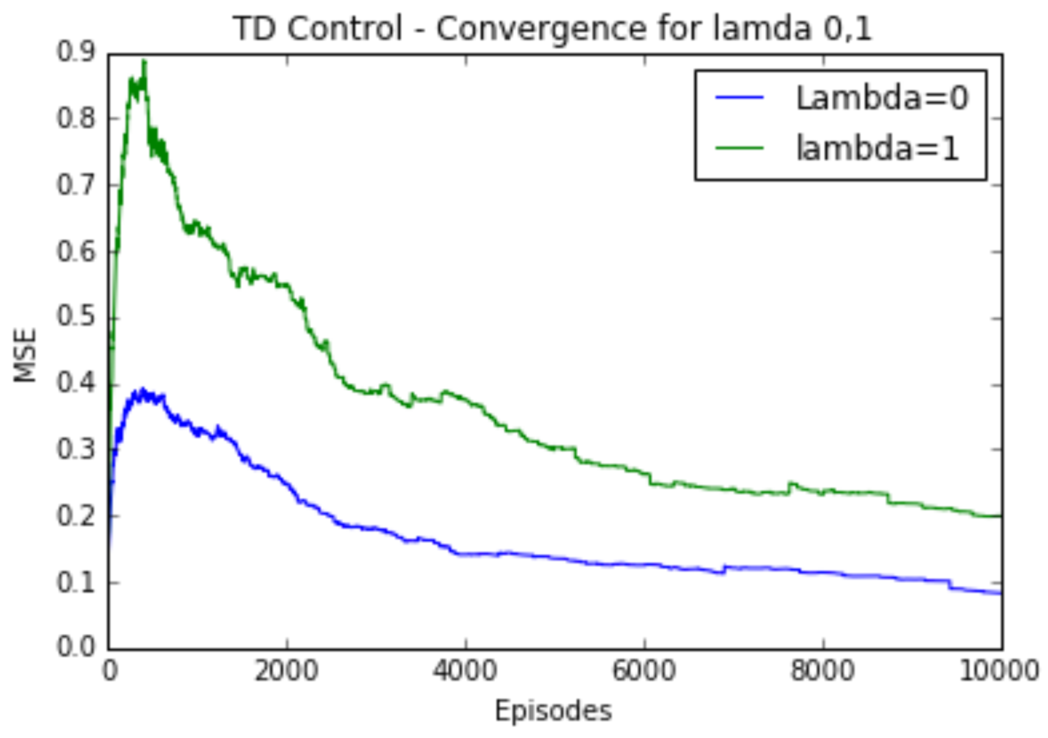
The code is run 10,000 times for each lambda value using the function td().



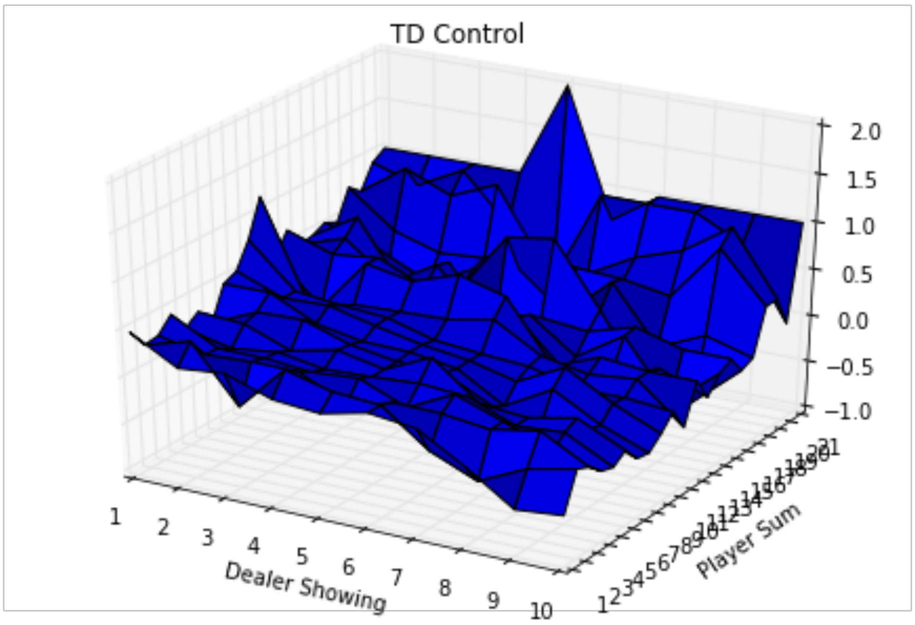
The figure above shows the final mean-squared error after 10,000 runs for lambda varying between zero and one with step sizes of 0.1.

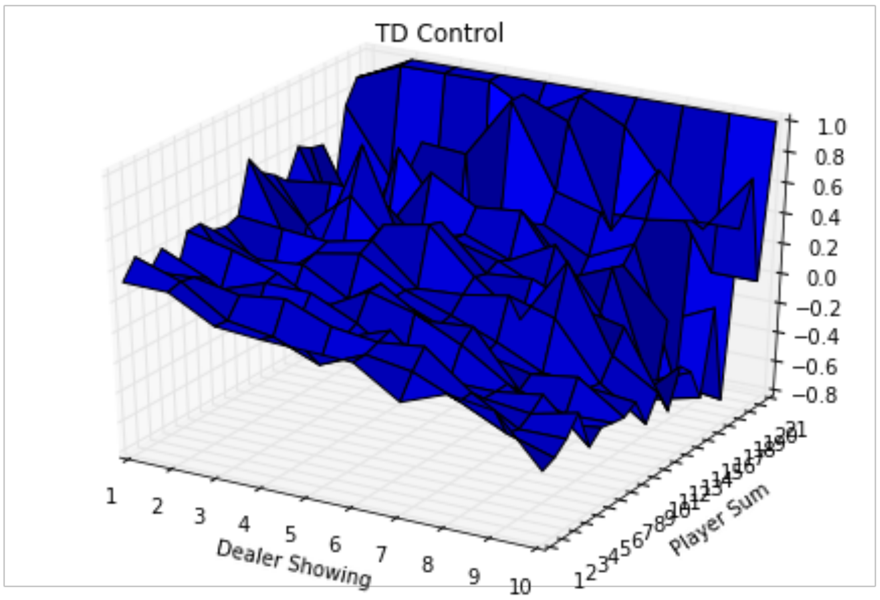
Note that this graph varies somewhat but typically a value of lambda close to 1 had a higher mean squared error. One could of course run this many times and average to get a better estimate of how lambda varies with MSE.

The question also asks for a plot of the mean squared error through time for lambda equal to zero and one only. This is given below, again lambda = 0, appeared to offer better convergence.



For completeness I also show the two value surfaces for Sarsa after 10,000 runs. This first for lambda equal to 1, the second for lambda equal to 0.





The surfaces are of course quite different compared to Monte-Carlo control. Many more episodes are required before the surface smooths out (I tried between 100,000 and 1m episodes and it did begin to look very much like the picture from Monte-Carlo).

Question 4

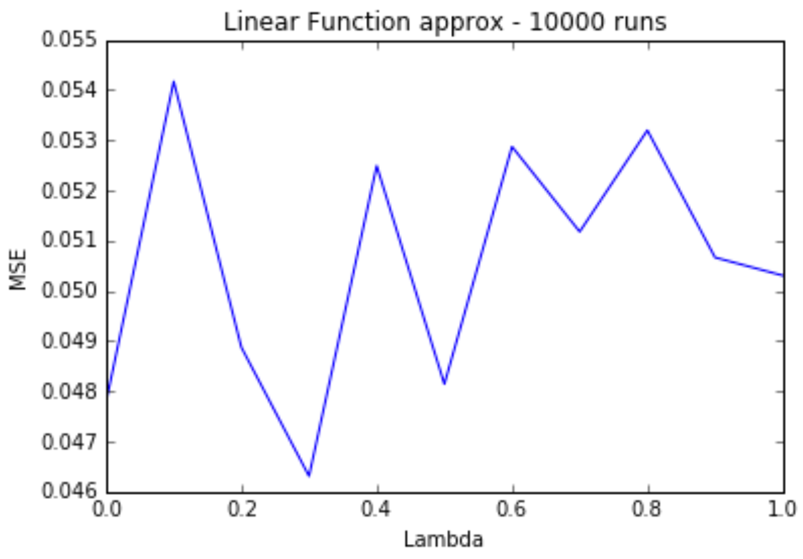
Use coarse coding with 36 features and repeat the Sarsa experiment using constant exploration of 0.1 and a constant step size with a linear function approximation.

Question 4 is implemented using the functions phi(), choose_action(), new_theta(), get_qvals() with the bulk of the work being done in lin_func_approx().

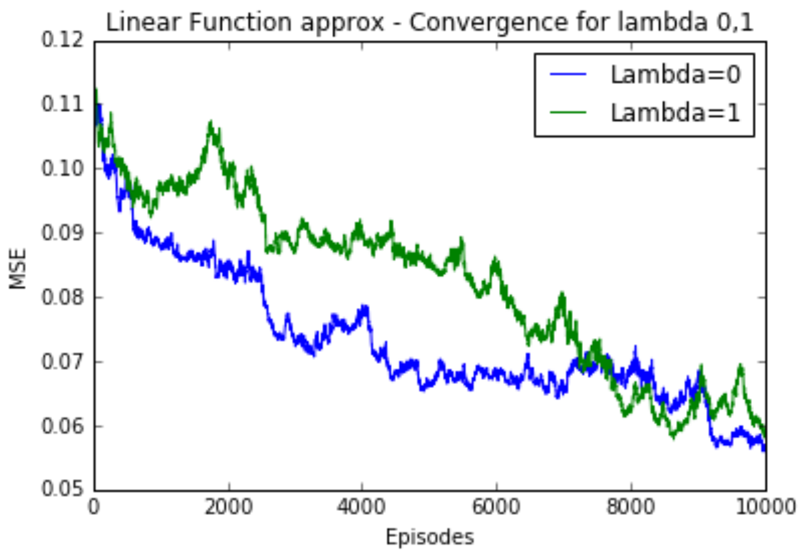
The key difference in this exercise is of course the linear function approximation. In the previous examples we tabulated the whole state-space. This is impossible in many environments and so a function approximation is necessary.

In this particular case we use coarse coding, which consists of over-lapping intervals covering the dealer, sum-total and actions with fewer dimensions, and a list of thetas to parameterise all of our features (in this case 36 of them).

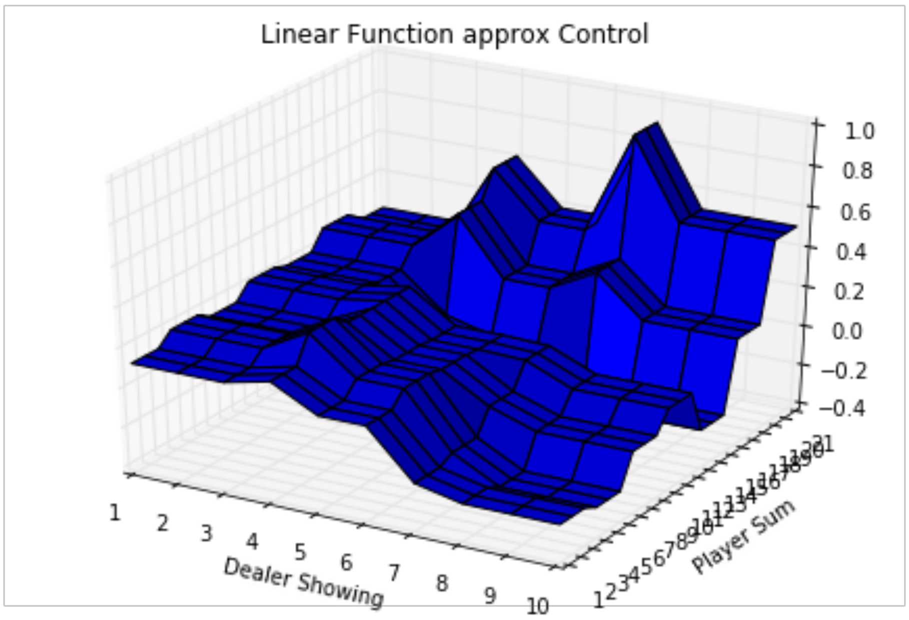
Again we run 10,000 episodes and are asked to provide the mean squared error against lambda. In this case we don'

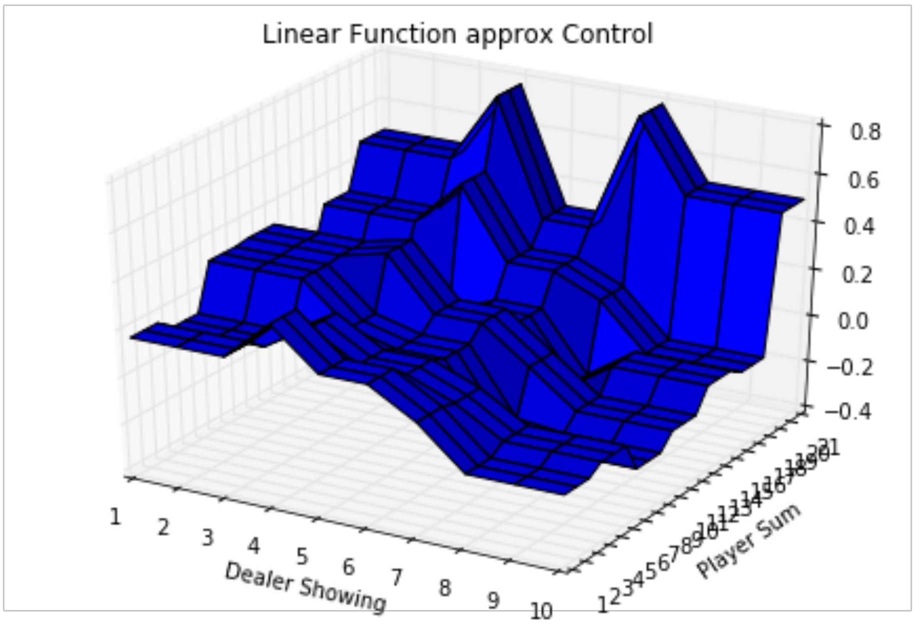


The mean squared error by episode for lambda equal to zero and one is given below. Again this will vary each time one runs the code.



For completeness I have also given the value surfaces for linear function approximation for lambda equal to zero and one. One can clearly see the impact of the coarse feature representation.





Question 5

- What are the pros and cons of bootstrapping in Easy21?
  Theoretically non-bootstrapping methods achieve a lower asymptotic error than bootstrapping errors, however in practise bootstrapping methods often outperform. In the case of Easy21 though, there are very few states and it is both quick and easy to simply do a full Monte-Carlo simulation millions of times, so in this limited example I see no particular advantage to bootstrapping here.

- Would you expect bootstrapping to help more in Easy21 or blackjack?
  The full game of blackjack has more options over the simplified Easy21, also because one can have negative card values in Easy21 the episodes may be longer. Thus I expect a benefit to boot-strapping in this case. However, blackjack is to my belief also fairly small game and the whole state space could be encoded and simulated with Monte-Carlo quite quickly. So in practise with modern computers I believe both methods are feasible but moving more towards a benefit for bootstrapping.

- What are the pros and cons of function approximation in Easy21?
  Well the cons are that there are more states than components to our theta vector in this case so depending upon the number of features using linear approximation will inevitably have errors. The pros are that we no longer need to represent that whole state-action feature space. (Again with a toy example like Easy21 this is not so onerous but in the case of large state-action spaces, such as GO then we can no longer tabulate the whole space so function approximation is essential).

- How would you modify the function approximator to get better results?
  Adding more features would improve the function approximator. Again in this toy example we could add enough that we could represent the whole space! Making the coarse coding more dense doesn't really cost any resources here. Clearly neural nets are a natural fit with reinforcement learning – to be specific under certain conditions an artificial neural net with more than one layer is a universal function approximator and we can thus learn highly complex and non-linear value functions. Possibly overkill for Easy21 though !