Bits, Brain & Behaviour - Coursework 2

Question 1: Monte-Carlo Control in Easy21

a For Monte-Carlo (MC) Control, on-policy ϵ-greedy first-visit (FV) MC Iterative Learning Algorithm is used in MATLAB. FV MC is used instead of every-visit (EV) MC because, according to Singh and Sutton (1996) *Reinforcement Learning with Replacing Eligibility Traces*, for large numbers of trials the FV estimate has a lower mean-squared error (MSE) than EV estimate (Lower variance and also unbiased estimator). This method also uses running means instead of batch averaging, without the need to store lists of returns for each state-action pair (s,a) thus requires less memory. Algorithm also avoids the unlikely assumption of exploring starts, since starting an episode with player sum > 10 is not realistic. Algorithm used obtains the Return (R) from the first appearance of a state-action pair (s,a) in the generated trace, which is basically final reward (r_{final}) since $\gamma = 1$, and updates the state-action value function estimate $\hat{Q}(s,a)$ using Equation 1 (online-averaging).

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha[r_{final} - \hat{Q}(s,a)] \text{ , } where\ \alpha = 1/N(s,a) \tag{1}$$

The step-size α is the reciprocal of cummulative number of visits to (s,a), N(s,a). Since process is non-stationary, it is useful to gradually decrease the sensitivity to recent return values. Step size chosen also satisfies the Robbins-Monroe condition.The ϵ-greedy policy used (Equation 2) satisfies GLIE criterion, as it asymptotically converges to the deterministic optimal policy, ensuring infinite exploration. The constant N_0, empirically chosen to be 100 (Fig.1), determines the ratio of exploration and exploitation done by the algorithm (See Part b). A high value of N_0 allows high exploration (low exploitation) since when ϵ is closer to 1 each action becomes equiprobable, but algorithm takes longer to converge. Algorithm was tested on 1,000,000 episodes due to computational power limitations. Also, if $\hat{Q}(s,a_1) = \hat{Q}(s,a_2)$, then $\pi(s,a) = 0.5$ for both.

$$\pi(s,a) = \begin{cases} 1 - \frac{\epsilon}{2} & , if\ a^* = \arg\max_a Q(s,a) \\ \frac{\epsilon}{2} & , if\ a \neq a^* \end{cases} \text{ , } where\ \epsilon = \frac{N_0}{N_0 + N(s)}. \tag{2}$$

b

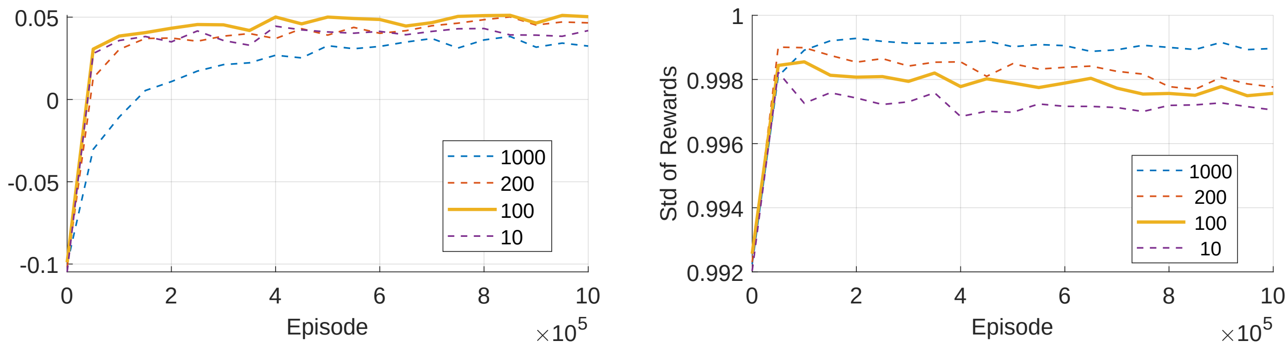

Figure 1: Learning Curve for MC Control showing mean and standard deviation of rewards vs episodes. Plot shows the learning curves for different values of N_0 (Solid line for chosen N_0 value).

Learning Curves were obtained by using validation sets of simulations, since plots of rewards vs episodes contain no useful information. Ideally after each training step, training temporarily stops, current policy is applied to several numbers of episodes of the game (validation set) and performance of system at this stage is estimated. Due to computational power limitations, and using 1,000,000 episodes, the validation step was performed after every 20,000 episodes for a validation set size of 100,000 episodes. Size of validation set was chosen such that that any variation in mean rewards would be smaller than changes in mean value itself, providing clearer trends. As expected, this approach makes the training longer, however it is a clean and methodologically appropriate way of obtaining such learning curves. Also does not have the issue of determining appropriate window size in methods involving sliding windows. Fig.1 shows that for $N_0 = 100$ the value of mean rewards converges to highest value and std of rewards is low after the end of process. This is because lower N_0 values converge faster with low exploration, thus lower mean values, and higher N_0 values do not fully converge.

(c) Fig.2 shows that for player sum values close to 21, the value function is maximum. It also shows that the expected returns of high dealer values are much lower than those for lower dealer values, regardless of player's sum. Local max at player sum of 11 is due to the fact that player cannot go bust at this value. Moreover, it should be noted that plot of value function for MC Control is considerably noisy.

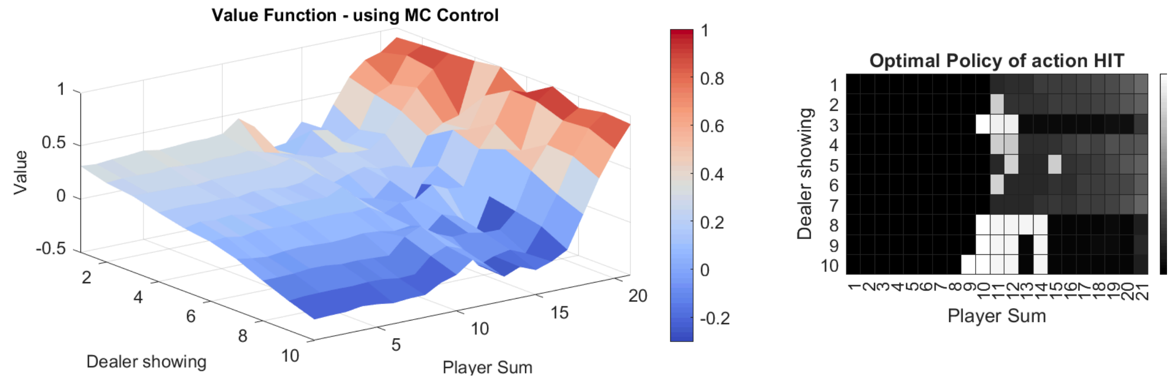

Figure 2: (Left) Optimal value function $V^*(s) = max_a Q^*(s,a)$ for MC Control. (Right) Optimal policy for the action "hit", using MC Control after 1,000,000 episodes ($N_0 = 100$).

Question 2: TD Learning (SARSA) in Easy21

(a) For this question, on-policy learning TD control (SARSA) algorithm was implemented in MATLAB for 1,000,000 episodes. This method is based on bootstrapping instead of sampling of complete episodic traces like MC Control. Specifically the implemented algorithm performs bootstrap only one time-step ahead. In this method, the update rule for the state-action value function estimate $\hat{Q}(s,a)$ is given by Equation 3. The update rule depends on the current state (S), the action chosen from S (A), the reward (R) from choosing A, the next state S' and the next action (A') chosen in the new state.

$$\hat{Q}(S,A) \leftarrow \hat{Q}(S,A) + \alpha[R + \hat{Q}(S',A') - \hat{Q}(S,A)] \ , \ where\ \alpha = 1/N(S,A) \tag{3}$$

Similarly as for MC Control, the step-size α is gradually decreased with visits to (S,A), thus satisfying the Robbins-Monroe condition. Algorithm also uses an ϵ-greedy policy derived from $\hat{Q}$ values, for choosing the next action, given by Equation 2, satisfying the GLIE criterion. Both ϵ and α thus ensure convergence to optimal solutions. By performing similar analysis for the most appropriate value of N_0 to be used, as was done in Q1 part b), this value was found to be 100. This value causes mean reward to converge to highest value, with relatively low standard deviation. Graphs are shown in Appendix A, fig.8.

(b) The learning curve showing mean and standard deviation of rewards was obtained using validation sets of simulations, which is described in Q.1 b). Again, 1,000,000 episodes were used and the validation step was performed after every 20,000 episodes for a validation set size of 100,000 episodes. To show the effect of step-size α on the curves, the learning curves were plotted (Fig.3) using the time-varying scalar step-size $\alpha_t = 1/(N(s_t, a_t)$ described in part a), and a range of constant values of α. Figure shows that the mean value of rewards converges to the highest value when step-size is time-varying. All constant α's do not ensure convergence to the optimal solution. This is because a constant value of α close to 1, means high learning rate but in the expense of convergence to wrong optimal solutions (instability). Graphs of large α's also have high-amplitude ripples. For constant α close to 0, the rate of convergence is much slower but with smaller ripples, so these do not converge in the 1,000,000 episodes used.

(c) Value function plot (Fig.4) was obtained using the time-varying scalar step-size, and $N_0 = 100$. Plot shows the same basic features as that for MC Control, however, in this case the plot of value function is less noisy (smoother).

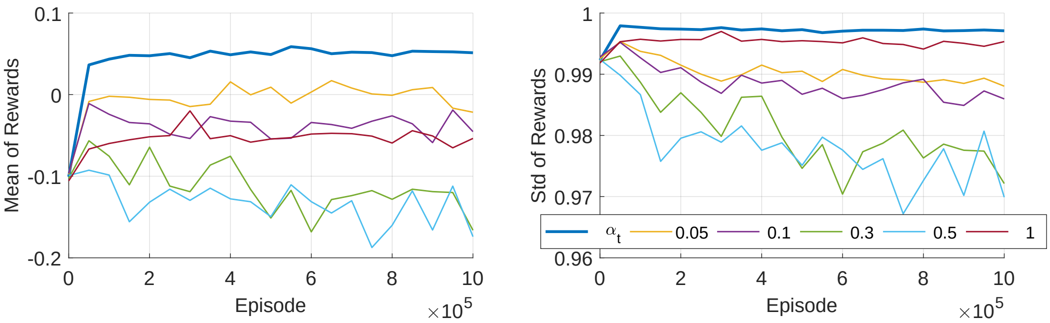

Figure 3: Dependency of Learning Curve on the strategy for handling α. Thicker line on plots belongs to the time-varying step-size.

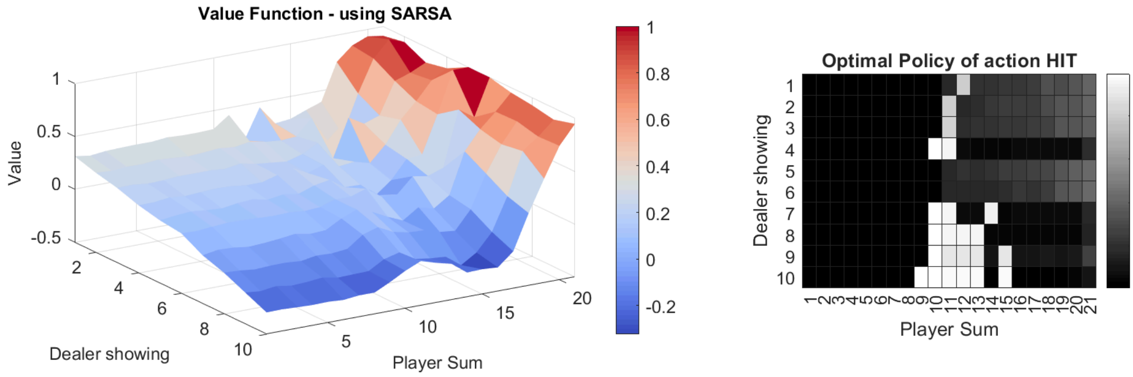

Figure 4: (Left) Optimal value function $V^*(s) = max_a Q^*(s,a)$ for SARSA. (Right) Optimal policy for the action "hit", using SARSA after 1,000,000 episodes ($N_0 = 100$).

Question 3: Q Learning in Easy21

(a) Q Learning algorithm implemented in MATLAB is an off-policy TD Control algorithm, where the next action (a') is chosen using the behaviour policy, while updating action value estimate $\hat{Q}(s,a)$ in direction of the alternative (better) action dictated by the target policy (deterministic). This is the difference to SARSA. Update rule is shown in equation 4.

$$\hat{Q}(S,A) \leftarrow \hat{Q}(S,A) + \alpha[R + \max_{a'} \hat{Q}(s',a') - \hat{Q}(S,A)] \ , \ where\ \alpha = 1/N(S,A) \tag{4}$$

Target Policy is greedy with respect to $\hat{Q}(s,a)$, while behaviour policy is ϵ-greedy with respect to $\hat{Q}(s,a)$. The step-size α used is again gradually decreased with visits to (S,A), satisfying the Robbins-Monroe condition, and the ϵ-greedy behaviour policy is given by Equation 2, satisfying the GLIE criterion, ensuring convergence. By performing similar analysis for the most appropriate value of N_0 to be used, as was done in Q1 part b), this value was found again to be 100. This value causes mean reward to converge to highest value, with relatively low standard deviation. Graphs are shown in Appendix A, fig.9.

(b) The learning curve showing mean and standard deviation of rewards was obtained using validation sets of simulations, which is described in Q.1 b). Again, 1,000,000 episodes were used and the validation step was performed after every 20,000 episodes for a validation set size of 100,000 episodes. To show the effect of ϵ-greediness on the curves, the learning curves were plotted (Fig.5) using the time-varying ϵ_t described in equation 2,and a range of constant values of ϵ. Generally, the higher the value of ϵ, the more the exploration done and the less the exploitation. Constant ϵ does not satisfy the GLIE criterion. The time-varying ϵ_t allows high exploration initially and gradually increases exploitation of agent. After ∞ episodes, exploration will reduce to zero. Fig.5 shows that as ϵ gets extremely close to 1 or 0, the mean value does not converge to optimal value. However, small values of ϵ converge to the optimal value in addition to ϵ_t, but only ϵ_t has the lowest variance. This is because N(s) in ϵ_t equation has not yet increased to infinity and is simply a small value that is gradually decreasing.

(c) Value function plot (Fig.6) was obtained using the time-varying ϵ and N_0=100. Plot shows the same basic features as that for MC Control and SARSA, however, in this case the plot shows only smooth changes in the value functions of neighbouring states (smoothest plot).

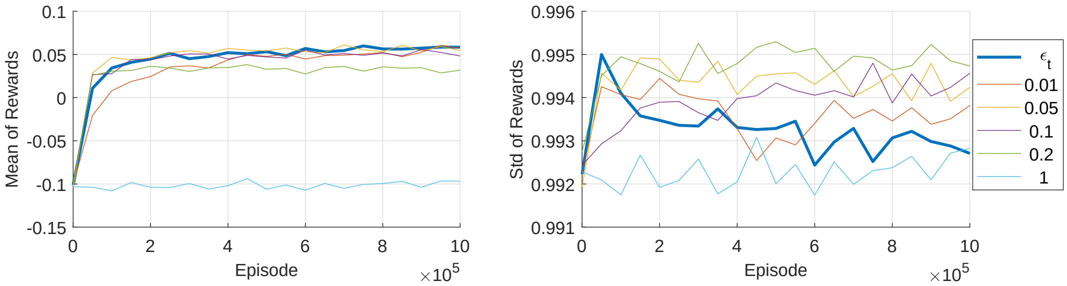

Figure 5: Dependency of Learning Curve on the strategy for handling ϵ-greediness. Thicker line on plots belongs to the time-varying ϵ_t.

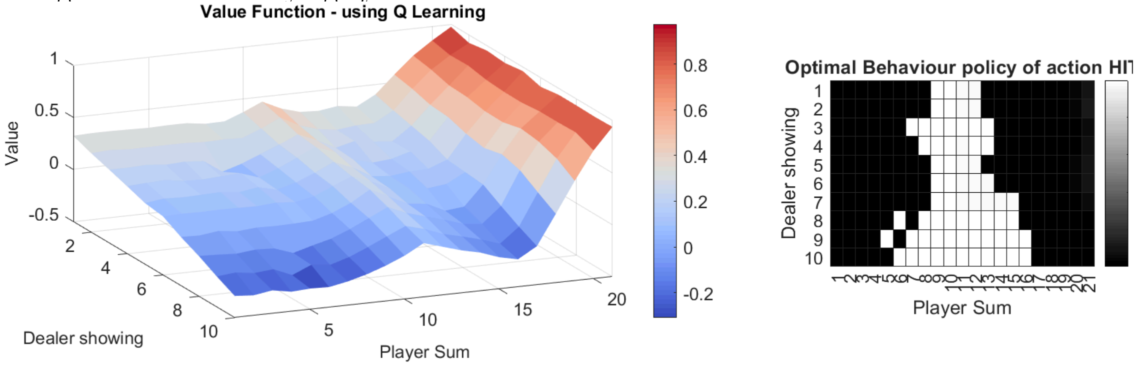

Figure 6: (Left) Optimal value function $V^*(s) = max_a Q^*(s,a)$ for Q-learning. (Right) Optimal Behavioural policy for the action "hit", using Q-learning after 1,000,000 episodes ($N_0 = 100$).

Question 4: Compare the algorithms

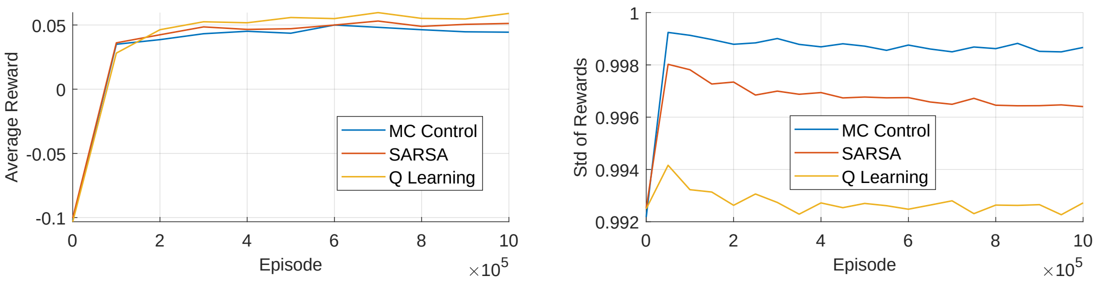

Figure 7: Learning curves of the three implementations. For mean plot, resolution is increased to identify differences. (validation steps=20,000 episodes, validation set size= 100,000 episodes)

The curves of mean rewards and standard deviation (std) of rewards (Fig.7) are equivalent to the changes that occur in the expected value and variance of value function estimate, respectively. Fig.7 shows that Q-Learning and SARSA (TD methods) outperform MC Control as they converge to higher mean reward value, while having a lower std of rewards. This can be explained by the Bias-Variance tradeoff of TD and MC methods. Both methods follow idea of generalised policy iteration (GPI). MC Control gives an unbiased estimate of Q(s) during policy evaluation, since update of $\hat{Q}(s)$ (Eq.1) is based on sample returns, but introduces some bias during policy improvement. TD Control methods give biased estimates in both steps of GPI, since updates of $\hat{Q}(s)$ (Eq.3 and 4) are based on other $\hat{Q}(s)$ estimates and are sensitive to initial estimate values. MC Control estimates have high variance since sample returns are the sum of many random variables (i.e. rewards of each step), while TD estimates' update rule contains only a fixed number of three random variables, yielding proportionally less variance than MC estimates. The better performance of SARSA and Q-learning compared to MC Control, is due to the fact that TD methods exploit the Markov Property of process, have lower convergence time and probably have a lower MSE ($MSE = Bias^2 + Variance$), i.e. a better balance of bias and variance of estimates. Both curves also show that Q-Learning outperforms SARSA in mean and std. The only difference between update rules of the two methods, is the max() function in Q-Learning, which always considers the best possible outcome by directly learning the optimal policy,while choosing actions from the exploring ϵ-greedy behavior policy. SARSA, takes a more conservative approach avoiding optimal paths with high risk of large cost. Since we have a low-cost environment, Q-learning yields higher average rewards, with low std.

Appendix A - Learning curves of SARSA and Q-Learning for different N_0 values.

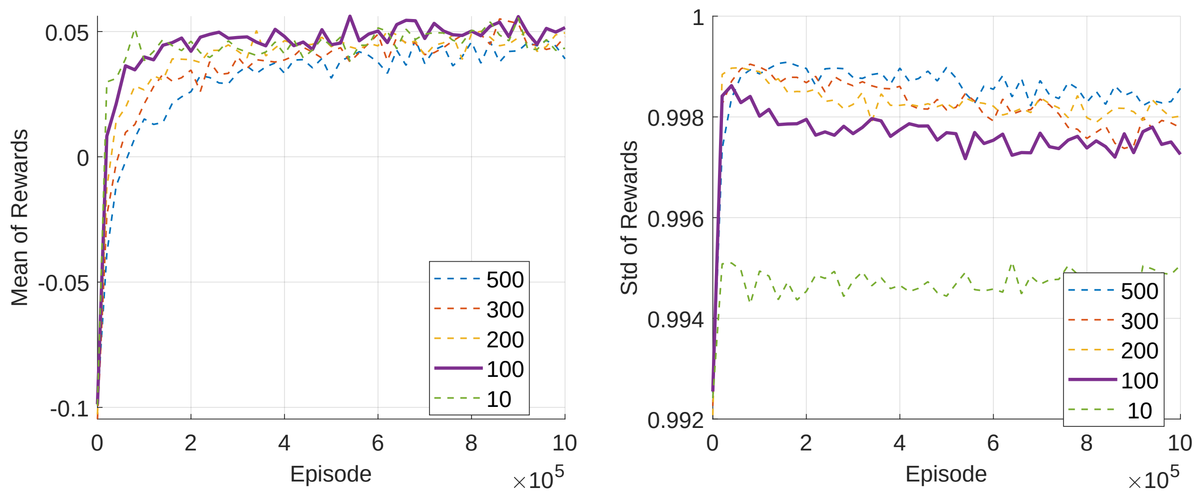

Figure 8: Learning curves of SARSA using different values of N_0. Number of episodes=1,000,000. Thick Line represents the chosen N_0 value.

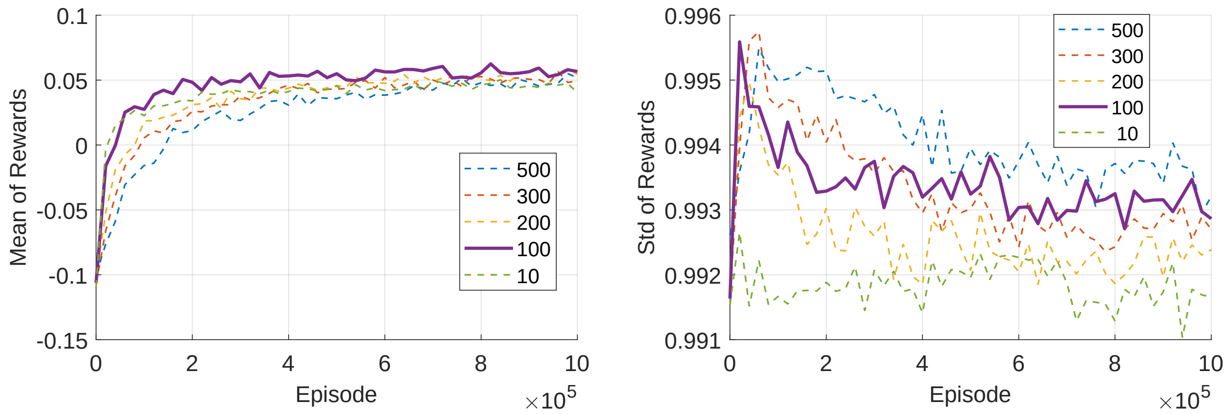

Figure 9: Learning curves of Q-learning using different values of N_0. Number of episodes=1,000,000. Thick Line represents the chosen N_0 value.

Appendix B - Matlab code used for implementation of Easy21

```
classdef Easy21_v3 < handle
   properties
        max_length;
        player_1stCard_val;
        dealer_1stCard_val;
        player_sum;
        dealer_sum;

        state;
        player_goes_bust;
        dealer_goes_bust;

        player_card_val;
        player_card_col;

        dealer_card_val;
        dealer_card_col;

        ret;
        terminal;
        t;
   end

   methods
        function init(obj,max_len)
            obj.max_length = max_len;
        end

        function state = reset(obj)
            obj.player_1stCard_val = randi([1 10]);
            obj.dealer_1stCard_val = randi([1 10]);

            obj.player_sum = obj.player_1stCard_val;
            obj.dealer_sum = obj.dealer_1stCard_val;

            obj.state = [obj.dealer_1stCard_val,obj.player_1stCard_val];

            obj.player_goes_bust=false;
            obj.dealer_goes_bust=false;

            obj.ret=0;
            obj.terminal = false;
            obj.t=0;

            state = obj.state;
        end

        function [state,Terminal,reward,Dealer_sum,Player_sum]=step(obj, ...
           action)
            %Action: 1 = hit , 0=stick
            %colour: 1=black , -1=red

            r=0; %Reward

            previous_player_sum = obj.player_sum;

            if action==1 %When player hits
                obj.player_card_val = randi([1 10]);

```

```
            x=rand; %Choosing random colour
            if x<(1/3)
                colour=-1;
            else
                colour=1;
            end
            obj.player_card_col=colour;

            obj.player_sum = obj.player_sum + (obj.player_card_val * ...
                obj.player_card_col);
            obj.player_goes_bust = (obj.player_sum>21 || ...
                obj.player_sum<1);

            if obj.player_goes_bust == 1
                r = -1;
                obj.terminal = true;
            end
        end

        if (action==0)&&(obj.terminal==0)%when player sticks
            while (obj.dealer_sum>0)&&(obj.dealer_sum<17)
                obj.dealer_card_val = randi([1 10]);
                z=rand; %Choosing random colour
                if z<(1/3)
                    col=-1;
                else
                    col=1;
                end
                obj.dealer_card_col=col;

                obj.dealer_sum = obj.dealer_sum + ...
                    (obj.dealer_card_val * obj.dealer_card_col);
                obj.dealer_goes_bust = (obj.dealer_sum>21 || ...
                    obj.dealer_sum<1);

                if obj.dealer_goes_bust == 1
                    r = 1;
                    obj.terminal = true;
                end
            end



            obj.terminal = true; %game ends when dealer sticks
            if ¬obj.dealer_goes_bust
                if obj.player_sum>obj.dealer_sum
                    r=1;
                elseif obj.player_sum<obj.dealer_sum
                    r=-1;
                elseif obj.player_sum==obj.dealer_sum
                    r=0;
                end
            end
        end

        obj.t = obj.t + 1;
        obj.ret = obj.ret + r;

        if ((obj.terminal==0) && (obj.t==obj.max_length))
            obj.terminal=true;
            if obj.player_sum>obj.dealer_sum
```

```
                    r=1;
                elseif obj.player_sum<obj.dealer_sum
                    r=-1;
                elseif obj.player_sum==obj.dealer_sum
                    r=0;
                end
            end

            if (obj.terminal==1) %Game has finished
                Terminal = obj.terminal;
                reward=r;
                obj.state(2) = previous_player_sum;
                state = obj.state;
            end

            if (obj.terminal==0)
                obj.state(2) = obj.player_sum;
                state = obj.state;
                reward=r;
                Terminal = obj.terminal;
            end

            Dealer_sum = obj.dealer_sum;
            Player_sum = obj.player_sum;
        end

   end
end
```

```
clc
close all
clear all

%% Defining the Game

%Calling the object into this file
import Easy21_v3
Game = Easy21_v3;

max_length = 1000;
init(Game,max_length);
state(1,:) = reset(Game);
Terminal=0;

index=1;

while Terminal==0
        x=rand;
        if x<0.75
            action=1;
        else
            action=0;
        end
    actions(index)=action;
    [next_state,Terminal,reward,s_D,s_P]=step(Game, action);
    Rew(index)=reward;
    index=index+1;
    if ¬Terminal
        state(index,:) = next_state;
    end
end
```

Appendix C - Matlab code used for MC Control of Easy21

```
function [policy,action] = MC_get_action(Q,policy,state,No,count_state)

    [ ¬ , action_greedy] = max(Q(state(1),state(2),:));
    if (Q(state(1),state(2),1)==Q(state(1),state(2),2))
        action = randi([1 2])-1;
        policy(state(1),state(2),1)=0.5;
        policy(state(1),state(2),2)=0.5;
    else
        action_greedy=action_greedy-1;
        epsilon = No/(No+(count_state(state(1),state(2))));
        %Choosing the action
        p=rand;
        if p<(1-epsilon/2) %This was deduced from initial form since ...
            number of possible actions =2
           action=action_greedy;
        else
           action=1-action_greedy;
        end

        if action_greedy==0
            policy(state(1),state(2),1)=(1-epsilon/2);
            policy(state(1),state(2),2)=epsilon/2;
        elseif action_greedy==1
            policy(state(1),state(2),2)=(1-epsilon/2);
            policy(state(1),state(2),1)=epsilon/2;
        end
    end
end
```

```
function [Q,Ret_episode,upd_policy,Ret,count_s,count_s_a]= ...
    MC_FV_episode(obj,No,init_s,Q,Ret,policy,count_s,count_s_a)

    states=[];
    actions =[];
    rewards=[];
    history=[];
    s=init_s;
    states(1,:)=s;
    index=1;

    %Generating the episode
    while true
        %This is basically N(s)
        count_s(states(index,1),states(index,2)) = ...
            count_s(states(index,1),states(index,2)) + 1; %This increments ...
            the visits to state s(1) and s(2)
        [upd_policy,action] = ...
            MC_get_action(Q,policy,states(index,:),No,count_s);
        state_action = [states(index,:),action];

        %The following are used to check if it is the first visit
        %of state-action pair (s,a)
        if index==1
            M=0;
        else
            M=ismember(state_action,history,'rows');
        end

```

```
        if ¬M
            history=[history;states(index,:),action]; %Stores first ...
                occurence of (s,a)
            count_s_a(states(index,1),states(index,2),action+1) = ...
                count_s_a(states(index,1),states(index,2),action+1) + 1;
        end

        actions=[actions;action];
        state_action = [states(index,:),action];
        [next_state,Terminal,reward,¬,¬]=step(obj, action);

        rewards=[rewards;reward];

        if Terminal
            break
        else
            states=[states;next_state];
        end
        index=index+1;
    end

    Return = sum(rewards);
    Ret_episode = Return;

    %The rest has to be done for the history of states

    for k=1:size(history,1)

        alpha=(1/count_s_a(states(k,1),history(k,2),history(k,3)+1));
        Ret(history(k,1),history(k,2),history(k,3)+1)=Return;

        Q(history(k,1),history(k,2),history(k,3)+1) = ...
        Q(history(k,1),history(k,2),history(k,3)+1)+ alpha*...
        (Ret(history(k,1),history(k,2),history(k,3)+1) - ...
            Q(history(k,1),history(k,2),history(k,3)+1));
    end
end
```

```
clc
close all
clear all

% Defining the Game

%Calling the object into this file
import Easy21_v3
Game = Easy21_v3;

No_vec=[1000 200 100 10]; %Testing the performance of MC Control with ...
    different No
%No_vec = 100; %Using the desired No value
for i=1:length(No_vec)

    No=No_vec(i);

    %Initialisation for MC

    Q_MC = zeros(10,21,2); %Q(s,a)
    Ret = zeros(10,21,2);
    count_s_a = zeros(10,21,2); %N(s,a)
    count_s = zeros(10,21); %N(s)
```

```
    max_length = 1000;
    n_episodes=1000000;
    run_mean=0;
    pol=0.5*ones(10,21,2); %Initial policy
    Rewards=zeros(1,n_episodes);

    index_test=50000;
    size_test_set=150000;
    Mean_vect=[];
    Std_vect=[];
    xaxis=[];

    %Generating episodes and calculating Q function
    for episode=1:n_episodes
        if mod((episode/n_episodes),0.05)==0
            episode/n_episodes
        end
        init(Game,max_length);
        init_s = reset(Game);
        [Q_MC,Return_episode,pol,Ret,count_s,count_s_a]=...
            MC_FV_episode(Game,No,init_s,Q_MC,Ret,pol,count_s,count_s_a);
        Rewards(episode)=Return_episode;

        %This is when validation testing is done
        if mod(episode,index_test)==0 || episode==1
            [mean_test,std_test]=test(pol,size_test_set);
            Mean_vect = [Mean_vect;mean_test];
            Std_vect = [Std_vect;std_test];
            xaxis=[xaxis;episode];
        end
    end

    figure(1)
    subplot(1,2,1)
    hold on
    grid on
    plot(xaxis,Mean_vect)
    ylabel('Mean of Rewards')
    xlabel('Episode')

    subplot(1,2,2)
    hold on
    grid on
    plot(xaxis,Std_vect)
    ylabel('Std of Rewards')
    xlabel('Episode')
end

legend(num2str([1000; 200; 100 ;10]))

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',14)
print(gcf, '-dpdf', 'Learning Curve .pdf','-fillpage')

%%
%Plot of policy of hitting
figure(2);
```

```
subplot(1,2,2)
h = heatmap(pol(:,:,2));
xlabel('Player Sum')
xlim([1 21])
ylabel('Dealer showing')
ylim([1 10])
title('Optimal Policy of action HIT')
colormap(gca,'gray')

%Histogram of Rewards
figure;
histogram(Rewards)

%% Storing the learning curves mean
avg_Rewards_MC=Mean_vect;
save('avg_Rewards_MC.mat','avg_Rewards_MC')

%% Storing the learning curves std
std_Rewards_MC=Std_vect;
save('std_Rewards_MC.mat','std_Rewards_MC')
%% Optimal value plot

[V_MC, index] = max(Q_MC,[],3);
index=index-1;

figure(2);
subplot(1,2,1)
surf(V_MC','EdgeColor','none')
ylabel('Player Sum')
ylim([1 21])
xlabel('Dealer showing')
xlim([1 10])
zlabel('Value')
title('Value Function - using MC Control')
colormap(gca,coolwarm)
hcb1=colorbar;

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',12)
print(gcf, '-dpdf', 'Value Function.pdf','-fillpage')
```

Appendix D - Matlab codes for SARSA Learning of Easy21

```
function [policy,action] = SARSA_get_action(Q,policy,state,No,count_state)

    [ ¬ , action_greedy] = max(Q(state(1),state(2),:));
    if (Q(state(1),state(2),1)==Q(state(1),state(2),2))
        action = randi([1 2])-1;
        policy(state(1),state(2),1)=0.5;
        policy(state(1),state(2),2)=0.5;
    else
        action_greedy=action_greedy-1;
        epsilon = No/(No+(count_state(state(1),state(2))));

        %Choosing the action
        p=rand;
        if p<(1-epsilon/2) %This was deduced from initial form since ...
            number of possible actions =2
           action=action_greedy;
        else
           action=1-action_greedy;
        end

        if action_greedy==0
            policy(state(1),state(2),1)=(1-epsilon/2);
            policy(state(1),state(2),2)=epsilon/2;
        elseif action_greedy==1
            policy(state(1),state(2),2)=(1-epsilon/2);
            policy(state(1),state(2),1)=epsilon/2;
        end
    end
end
```

```
function [Q,Return_episode,upd_pol,count_s,count_s_a]=...
    SARSA_episode(obj,No,initial_s,Q,pol,count_s,count_s_a,alpha_choice)

    s=initial_s;
    Terminal=false;
    [upd_pol,action] = SARSA_get_action(Q,pol,s,No,count_s);

    %Generating the episode

    while ¬Terminal

        %This is basically N(s) - used for step size (alpha)
        count_s(s(1),s(2)) = count_s(s(1),s(2)) + 1; %This increments the ...
            visits to state s(1) and s(2)
        count_s_a(s(1),s(2),action+1) = count_s_a(s(1),s(2),action+1) + 1;

        [next_state,Terminal,reward,¬,¬]=step(obj, action);

        %Obtain next action - ONLY IF NOT terminal


        if ¬Terminal
            [upd_pol,next_action] = ...
                SARSA_get_action(Q,upd_pol,next_state,No,count_s);
            d=Q(next_state(1),next_state(2),next_action+1);
        else
            d=0;
        end
```

```

        if alpha_choice==0 %Then we use time-varying step-size
            alpha=(1/count_s_a(s(1),s(2),action+1));
        else
            alpha=alpha_choice;
        end

        %Updating Q-value
        Q(s(1),s(2),action+1) = Q(s(1),s(2),action+1) + alpha*(reward + d ...
            - Q(s(1),s(2),action+1));
        if ¬Terminal
            s=next_state;
            action = next_action;
        else
            Return_episode = reward;
        end
    end
end
```

```
clc
close all
clear all

%Defining the Game

%Calling the object into this file
import Easy21_v3
Game = Easy21_v3;

%No_vec=[500 300 200 100 10]; %Testing the performance of MC Control with ...
    different No
No_vec = 100; %Using the desired No value
%alpha = [0 0.01 0.05 0.1 0.3 0.5 1];
alpha=0;

for i=1:length(alpha)

    No=No_vec;
    %Initialisation for SARSA

    Q_Sar = zeros(10,21,2); %Q(s,a)
    count_s_a = zeros(10,21,2); %N(s,a)
    count_s = zeros(10,21); %N(s)

    max_length = 1000;
    n_episodes=1000000;
    pol=0.5*ones(10,21,2);
    Rewards=zeros(1,n_episodes);

    index_test=50000;
    size_test_set=100000;
    Mean_vect=[];
    Std_vect=[];
    xaxis=[];

    for episode=1:n_episodes
        if mod((episode/n_episodes),0.05)==0
            episode/n_episodes
        end
        init(Game,max_length);
        init_s = reset(Game);
```

```
        [Q_Sar,Return_episode,pol,count_s,count_s_action]=...
         SARSA_episode(Game,No,init_s,Q_Sar,pol,count_s,count_s_a,alpha(i));
        Rewards(episode)=Return_episode;

        %This is when validation testing is done
            if mod(episode,index_test)==0 || episode==1
                [mean_test,std_test]=test(pol,size_test_set);
                Mean_vect = [Mean_vect;mean_test];
                Std_vect = [Std_vect;std_test];
                xaxis=[xaxis;episode];
            end
    end

    figure(1)
    subplot(1,2,1)
    hold on
    grid on
    plot(xaxis,Mean_vect)
    ylabel('Mean of Rewards')
    xlabel('Episode')

    subplot(1,2,2)
    hold on
    grid on
    plot(xaxis,Std_vect)
    ylabel('Std of Rewards')
    xlabel('Episode')
end
%legend(num2str([0;0.01;0.05;0.1;0.3;0.5;1]))

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',14)
print(gcf, '-dpdf', 'Learning Curve .pdf','-fillpage')

%%
%Plot of policy of hitting
figure(2);
subplot(1,2,2)
h = heatmap(pol(:,:,2));
xlabel('Player Sum')
xlim([1 21])
ylabel('Dealer showing')
ylim([1 10])
title('Optimal Policy of action HIT')
colormap(gca,'gray')

%Histogram of Rewards
figure;
histogram(Rewards)

%% Storing Learning Curves mean
avg_Rewards_SARSA=Mean_vect;
save('avg_Rewards_SARSA.mat','avg_Rewards_SARSA')
%% Storing Learning Curves std
std_Rewards_SARSA=Std_vect;
save('std_Rewards_SARSA.mat','std_Rewards_SARSA')

%% Optimal value plot
```

```
[V_Sarsa, index] = max(Q_Sar,[],3);

figure(2);
subplot(1,2,1)
surf(V_Sarsa','EdgeColor','none')
ylabel('Player Sum')
ylim([1 21])
xlabel('Dealer showing')
xlim([1 10])
zlabel('Value')
title('Value Function - using SARSA')
colormap(gca,coolwarm)
hcb1=colorbar;

h=gcf;
set(h,'Position',[50 50 1100 700]);
set(h,'PaperOrientation','landscape');
print(gcf, '-dpdf', 'Value Function.pdf')

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',12)
print(gcf, '-dpdf', 'Value Function.pdf','-fillpage')
```

Appendix E - Matlab codes for Q-Learning of Easy21

```
function [policy,action] = ...
    Q_Learning_get_action(Q,policy,state,No,count_state,epsilon_choice)

    [ ¬ , action_greedy] = max(Q(state(1),state(2),:));
    if (Q(state(1),state(2),1)==Q(state(1),state(2),2))
        action = randi([1 2])-1;
        policy(state(1),state(2),1)=0.5;
        policy(state(1),state(2),2)=0.5;
    else
        action_greedy=action_greedy-1;

        if epsilon_choice==0
            epsilon = No/(No+(count_state(state(1),state(2))));
        else
            epsilon=epsilon_choice;
        end

        %Choosing the action
        p=rand;
        if p<(1-epsilon/2) %This was deduced from initial form since ...
            number of possible actions =2
           action=action_greedy;
        else
           action=1-action_greedy;
        end

        if action_greedy==0
            policy(state(1),state(2),1)=(1-epsilon/2);
            policy(state(1),state(2),2)=epsilon/2;
        elseif action_greedy==1
            policy(state(1),state(2),2)=(1-epsilon/2);
            policy(state(1),state(2),1)=epsilon/2;
        end
    end
end
```

```
function [Q,Ret_episode,upd_pol,opt_pol,count_s,count_s_a]=...
    Q_Learning_episode(obj,No,init_s,Q,pol,opt_pol,count_s,count_s_a,epsilon)

    s=init_s;
    Terminal=false;

    %Generating the episode

    while ¬Terminal
        %This is basically N(s) - used for step size (?)
        count_s(s(1),s(2)) = count_s(s(1),s(2)) + 1; %This increments the ...
            visits to state s(1) and s(2)

        [upd_pol,action] = Q_Learning_get_action(Q,pol,s,No,count_s,epsilon);
        count_s_a(s(1),s(2),action+1) = count_s_a(s(1),s(2),action+1) + 1;

        [next_state,Terminal,reward,¬,¬]=step(obj, action);

        %Obtain next action - ONLY IF NOT terminal


        if ¬Terminal
```

```
            [opt_pol,optimal_action] = ...
                Q_Learning_optimal_action(Q,opt_pol,next_state);
            d=Q(next_state(1),next_state(2),optimal_action+1);
        else
            d=0;
        end

        %Updating Q-value
        Q(s(1),s(2),action+1) = Q(s(1),s(2),action+1) + ...
            (1/count_s_a(s(1),s(2),action+1))*(reward + d - ...
            Q(s(1),s(2),action+1));

        if ¬Terminal
            s=next_state;
        else
            Ret_episode = reward;
        end
    end
end
```

```
clc
close all
clear all

% Defining the Game

%Calling the object into this file
import Easy21_v3
Game = Easy21_v3;

%No_vec=[500 300 200 100 10]; %Testing the performance of MC Control with ...
    different No
No_vec = 100; %Using the desired No value
e = [0,0.01,0.05,0.1,0.2,1]; %Epsilon-greedyness

for i=1:length(e)
    No=No_vec;

    %Initialisation for SARSA

    Q_QL = zeros(10,21,2); %Q(s,a)
    c_s_a = zeros(10,21,2); %N(s,a)
    c_s = zeros(10,21); %N(s)

    max_length = 1000;
    n_episodes=1000000;
    b_pol=0.5*ones(10,21,2); %Behavioral policy
    t_pol = 0.5*ones(10,21,2); %Target policy
    Rewards=zeros(1,n_episodes);

    index_test=50000;
    size_test_set=100000;
    Mean_vect=[];
    Std_vect=[];
    xaxis=[];

    for episode=1:n_episodes
        if mod((episode/n_episodes),0.05)==0
            episode/n_episodes
        end
        init(Game,max_length);
```

```
        init_s = reset(Game);
        [Q_QL,Ret_episode,b_pol,t_pol,c_s,c_s_a]=...
        Q_Learning_episode(Game,No,init_s,Q_QL,b_pol,t_pol,c_s,c_s_a,e(i));
        Rewards(episode)=Ret_episode;

        %This is when validation testing is done
        if mod(episode,index_test)==0 || episode==1
             [mean_test,std_test]=test(b_pol,size_test_set);
             Mean_vect = [Mean_vect;mean_test];
             Std_vect = [Std_vect;std_test];
             xaxis=[xaxis;episode];
        end
    end

    figure(1)
    subplot(1,2,1)
    hold on
    grid on
    plot(xaxis,Mean_vect)
    ylabel('Mean of Rewards')
    xlabel('Episode')

    subplot(1,2,2)
    hold on
    grid on
    plot(xaxis,Std_vect)
    ylabel('Std of Rewards')
    xlabel('Episode')
end

legend(num2str([0;0.01;0.05;0.1;0.2;1]))

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',12)
print(gcf, '-dpdf', 'Learning Curve .pdf','-fillpage')

%%
%Plot of policy of hitting
figure(2);
subplot(1,2,2)
h = heatmap(b_pol(:,:,2));
xlabel('Player Sum')
xlim([1 21])
ylabel('Dealer showing')
ylim([1 10])
title('Optimal Behaviour policy of action HIT')
colormap(gca,'gray')

%Histogram of Rewards
figure;
histogram(Rewards)

%% Storing Learning Curves mean
avg_Rewards_QLearning=Mean_vect;
save('avg_Rewards_QLearning.mat','avg_Rewards_QLearning')
%% Storing Learning Curves std
std_Rewards_QLearning=Std_vect;
save('std_Rewards_QLearning.mat','std_Rewards_QLearning')
```

```
%% Optimal value plot

[V_Q_learning, index] = max(Q_QL,[],3);

figure(2);
subplot(1,2,1)
surf(V_Q_learning','EdgeColor','none')
ylabel('Player Sum')
ylim([1 21])
xlabel('Dealer showing')
xlim([1 10])
zlabel('Value')
title('Value Function - using Q Learning')
colormap(gca,coolwarm)
hcb1=colorbar;

h=gcf;
set(h,'Position',[50 50 1100 700]);
set(h,'PaperOrientation','landscape');
print(gcf, '-dpdf', 'Value Function.pdf')

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',12)
print(gcf, '-dpdf', 'Value Function.pdf','-fillpage')
```

Appendix F - Matlab codes/functions used by all methods

```
function [mean_test std_test]=test(policy,size_test_set) %validation stage
    Reward=zeros(1,size_test_set);
    for i=1:size_test_set
        Terminal=0;
        s=[randi(10) randi(10)];

        while ¬Terminal
            p=rand;
            if p<policy(s(1),s(2),1)
                action=0;
            else
                action=1;
            end

            [s,Terminal,reward]=step(s, action);
        end

        Reward(i)=reward;
    end

    mean_test=mean(Reward);
    std_test=std(Reward);

end

function [state,Terminal,reward]=step(s, action)
            %Action: 1 = hit , 0=stick
            %colour: 1=black , -1=red

            r=0; %Reward

            player_sum = s(2);
            dealer_sum=s(1);

            if action==1 %When player hits
                player_card_val = randi([1 10]);

                x=rand; %Choosing random colour
                if x<(1/3)
                    colour=-1;
                else
                    colour=1;
                end
                player_card_col=colour;
                player_sum = player_sum + (player_card_val * ...
                   player_card_col);
                player_goes_bust = (player_sum>21 || player_sum<1);

                Terminal=0;

                if player_goes_bust == 1
                    r = -1;
                    Terminal = true;
                end


            end

            if (action==0)%when player sticks
```

```
                while (dealer_sum>0)&&(dealer_sum<17)
                    dealer_card_val = randi([1 10]);
                    z=rand; %Choosing random colour
                    if z<(1/3)
                        col=-1;
                    else
                        col=1;
                    end
                    dealer_card_col=col;

                    dealer_sum = dealer_sum + (dealer_card_val * ...
                       dealer_card_col);
                    dealer_goes_bust = (dealer_sum>21 || dealer_sum<1);

                    if dealer_goes_bust == 1
                        r = 1;
                    end
                end

                Terminal = true; %game ends when dealer sticks
                if ¬dealer_goes_bust
                    if player_sum>dealer_sum
                        r=1;
                    elseif player_sum<dealer_sum
                        r=-1;
                    elseif player_sum==dealer_sum
                        r=0;
                    end
                end
            end

            if (Terminal==1) %Game has finished
                reward=r;
                state = s;
            end

            if Terminal==0
                state(2) = player_sum;
                state(1) = s(1);
                reward=r;
            end
end
```

```
close all ; clear all ; clc

%Loading Learning Curves of the three methods
load('avg_Rewards_MC.mat');
load('std_Rewards_MC.mat');

load('avg_Rewards_SARSA.mat');
load('std_Rewards_SARSA.mat');

load('avg_Rewards_QLearning.mat');
load('std_Rewards_QLearning.mat');

%construction of the two x-axes to be used
n_episodes=1000000;
index_test_mean=100000;
index_test_std=50000;
xaxis_means=[];
xaxis_std =[];
```

```
for episode=1:n_episodes
    if mod(episode,50000)==0
        episode
    end


            if mod(episode,index_test_mean)==0 || episode==1
                xaxis_means=[xaxis_means;episode];
            end

            if mod(episode,index_test_std)==0 || episode==1
                xaxis_std=[xaxis_std;episode];
            end
end

%% Average Reward plots
figure(1);
subplot(1,2,1)
hold on
grid on
plot(xaxis_means,avg_Rewards_MC);
plot(xaxis_means,avg_Rewards_SARSA);
plot(xaxis_means,avg_Rewards_QLearning);
xlabel('Episode'); ylabel('Average Reward')
legend('MC Control','SARSA','Q Learning','Location','best')

%% Std Plots
figure(1);
subplot(1,2,2)
hold on
grid on
plot(xaxis_std,std_Rewards_MC);
plot(xaxis_std,std_Rewards_SARSA);
plot(xaxis_std,std_Rewards_QLearning);
xlabel('Episode'); ylabel('Std of Rewards')
legend('MC Control','SARSA','Q Learning','Location','best')
hold off

%%
h=gcf;
h.PaperPositionMode='auto';
set(h,'PaperOrientation','landscape');
set(findall(gcf,'-property','FontSize'),'FontSize',12)
print(gcf, '-dpdf', 'Value Function.pdf','-fillpage')
```