Institute for Aerospace Studies
UNIVERSITY OF TORONTO

Abhinav Grover (1001257418)
abhinav.grover@mail.utoronto.ca
March 24, 2020

Markov Decision Processes, Classical Reinforcement Learning,and Model Predictive Control

AER 1517: Assignment 2

1 Cliff World Problem

The purpose of this report is to document the result and finding from solving the Assignment 2. The section talks about solving a cliff-world environment (shown in figure 1) with model-based methods like generalized policy iteration (GPI) and linear-programming (LP), as well as model free methods like Monte-Carlo and Q-learning.

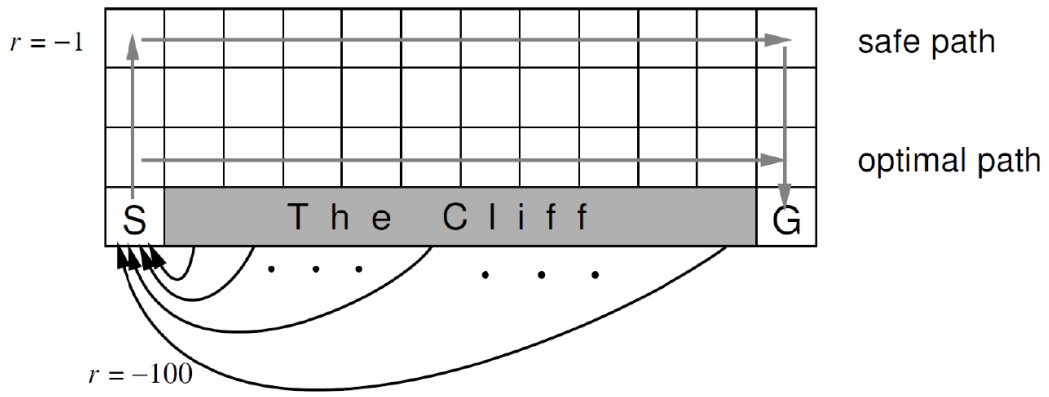

Figure 1: The cliff-world environment to be solved where S is the starting state and G is the goal state. The rewards are labelled with r.

1.1 Model-Based Methods

1.1.1 Generalized Policy Iteration

The generalized policy iteration algorithm has been implemented based on the lecture notes. There are two major steps involved in GPI: Policy Evaluation and Policy Improvement. GPI is named quite appropriately,

i.e., if the iterations of the policy evaluation step are constrained to 1, the algorithm will become value iteration (VI). If the ending condition for policy evaluation step is changed to depend on the convergence of the state-value function (limited by a maximum number of iterations), the algorithm will become standard policy iteration (PI). Figure 2 shows the resultant trajectory and state-value function heat-map of the two extremes of GPI. As it turns out, both PI and VI produce the same heat-map and final optimal trajectory. It is worth noting that the convergence time of PI at an average is higher than that of VI. This because, VI algorithm is guaranteed to converge in contrast to PI algorithm because of which PI seldom reaches maximum allowed iterations without convergence.

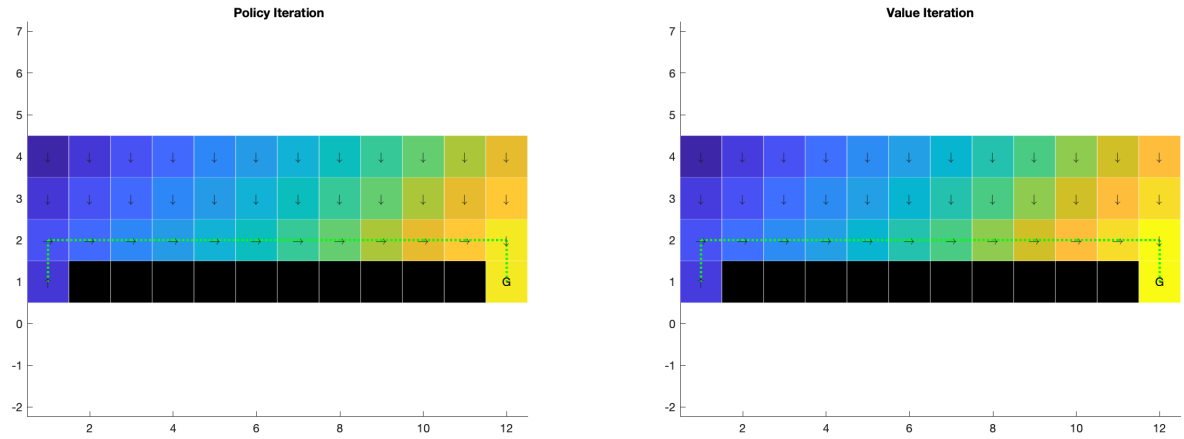

Figure 2: Cliff world traversal trajectory and the state value function heat-maps for both policy iteration (left) and value iteration (right).

1.1.2 Linear Programming (bonus)

The model-based control problem for the cliff-world can also be formulated as a linear-programming optimization problem, shown in equation 1. The resultant trajectory and state-value function heat-map is shown in figure 3.

$$\begin{aligned} \min_{\xi} \quad & \sum_{s} c(s)\xi(s) \\ s.t. \quad & \xi(s) \geq \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma\xi(s')], \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{1}$$

It can be proved that the solution to the linear program in 1 will be the optimal state-value function.

Since the inequality in 1 is required to be true for all actions in set $\mathcal{A}$, we can reduce the number of inequalities by taking the max of the right-hand side with respect to the action. Thus, the optimization in 1 is equivalent to 2. Now for a linear cost function with linear inequality constraints, the the boundary condition is optimal solution. This implies that at the optimal value, the inequality constraints should be fulfilled with equality if LHS and RHS. Thus, for the optimal $\xi^*(s)$ value, equation 3 will hold.

$$\begin{aligned} &\min_{\xi} \quad \sum_{s} c(s)\xi(s) \\ &s.t. \quad \xi(s) \geq \max_{a \in \mathcal{A}} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \xi(s')], \qquad \forall s \in \mathcal{S} \end{aligned} \tag{2}$$

$$\xi^*(s) = \max_{a \in \mathcal{A}} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \xi^*(s')], \qquad \forall s \in \mathcal{S} \tag{3}$$

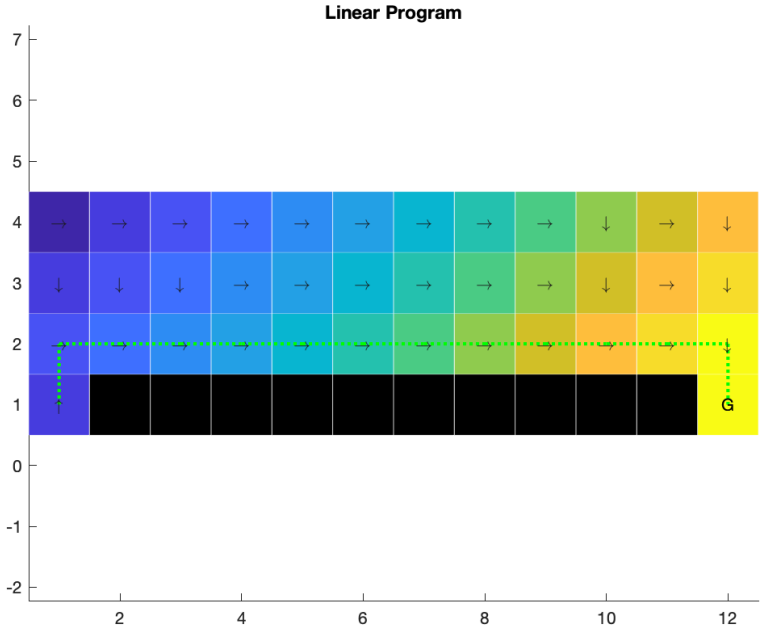

Figure 3: Cliff world traversal trajectory and the state value function heat-map for Linear Programming solution.

1.2 Sampling-Based Methods

Sampling-based methods are used for solving control problem when the system model is unknown. By running some episodes through the system/environment, the agent gathers samples of states, actions, and rewards for a given policy, which is then used to converge to an optimal policy.

1.2.1 Monte-Carlo

One way to gather samples from the environment is to use the Monte-Carlo method. During the policy evaluation step of GPI, an episode is run using a given policy in order to estimate the corresponding action value function $\mathcal{Q}(s, a)$.After that, the discounted cumulative reward is calculated for each state which is used to estimate the action-value function. This estimated action-value function is then used to improve the policy greedily. Figure 4 shows the resultant state-value heat-map and the optimal trajectory obtained by using the Mote-Carlo method. It is worth noting an ϵ-soft policy has been used simulate exploration of the environment. A high value of the exploration parameter ϵ with a corresponding reduction in the decay parameter value produced the result shown. With a constant ϵ, the algorithm fails to find an optimal path due to consistent under/over exploration. A sweep of the exploration parameter might be required to find one that solves the problem.

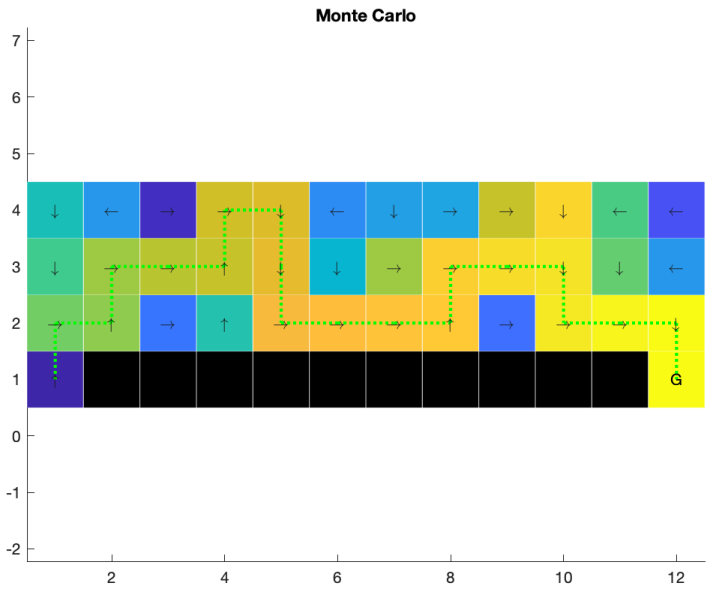

Figure 4: Cliff world traversal trajectory and the state value function heat-map for Monte-Carlo solution.

1.2.2 Q-learning

Q-learning is also a sampling-based learning method with similarities to Value Iteration. It is bootstrap method where Bellman equation is applied on every step of the episode. Figure 5 shows the resultant state-value function heat-maps and the resultant trajectory when using Q-learning with and without exploration decay. It is important to note that including exploration decay does not make a significant difference in the state-value heat-map or the policy .

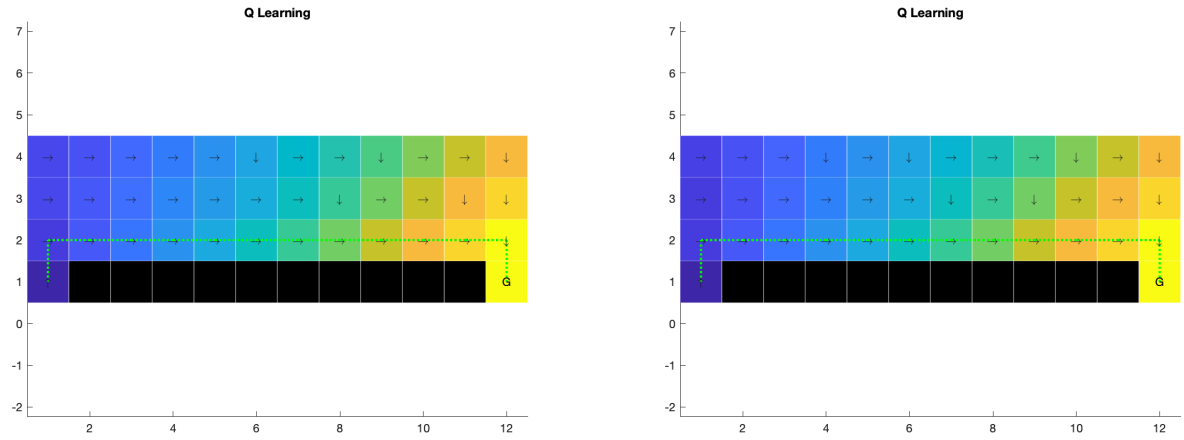

Figure 5: Cliff world traversal trajectory and the state value function heat-maps for Q-learning with (left) and without decay (right).

Figure 6 shows the results of running Q-learning with simulated measurement noise. The algorithms is able to handle the high noise and still able to produce the optimal policy.

When comparing the Monte-Carlo and Q-learning, it is clear that Q-learning produces a better and more optimal solution. It is also important to note that Q-learning takes less time to complete as compared to Monte-Carlo, though the difference is not significant.

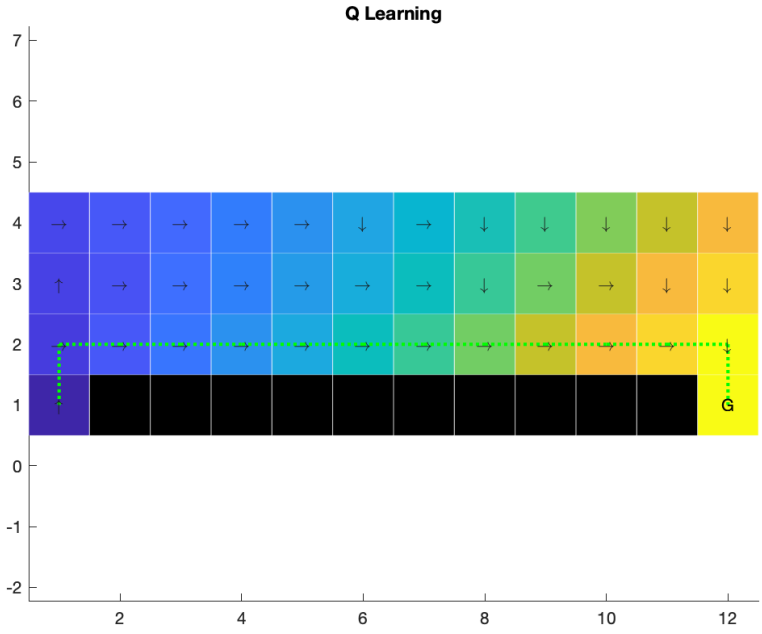

Figure 6: Cliff world traversal trajectory and the state value function heat-maps for Q-learning with high simulated noise in the environment ($\alpha = 0.2$).

2 Mountain Car Problem

This section talks about solving the classic mountain car problem, first proposed in [1], using model-based methods like GPI and model predictive control (MPC).

2.1 Generalized Policy Iteration

In order to use GPI on a continuous state and action space problem like mountain car, the state and action space needs to be discretized. In this report we use two techniques for this discretization process: nearest neighbour and linear interpolation.

2.1.1 Nearest Neighbour

This is a state-action discretization approach in which the discrete states-action pair closest to the continuous state are used as the proxy for the current state. The noise added to the motion model produces the stochasticity in state-transition model, that is, the transition model is not a one-hot deterministic model.

This stochastic transition model, with its corresponding reward structure, when used in a GPI setting produces the optimal state-value heat-map and trajectory shown in figure 7. After running some experiments, it is concluded that the model noise appears to have a significant affect on the quality of the solution.

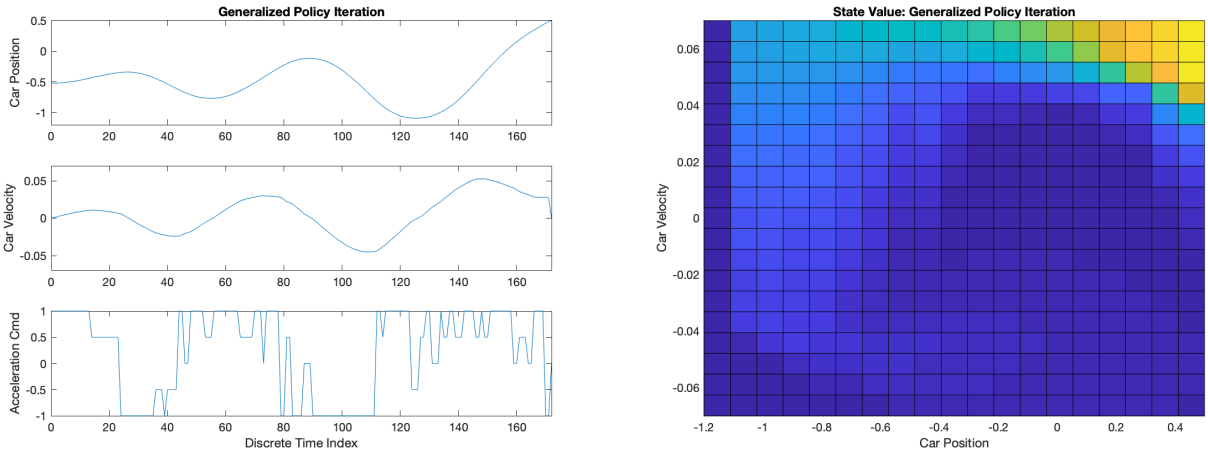

Figure 7: Mountain car trajectory (left) and the state value function heat-map (right)for running GPI on transition model discretized using the nearest neighbour method.

Intuitively, the optimal solution should have the car move back first to solve the problem in limited number of swings. GPI with nearest neighbour discretization does not appear to arrive to the optimal solution. This could be because of the low resolution of the discretization causing certain transition probabilities to inflate while other to be much lower than expected. Another reason could be the limited sample size of 50 used to estimate the prabobility distribution per state-action pair.

2.1.2 Linear Interpolation

Linear Interpolation is the second methods used to dicretize the transition model. Figure 8 shows the result of using GPI to solve the mountain car problem with a linearly interpolated transition model.

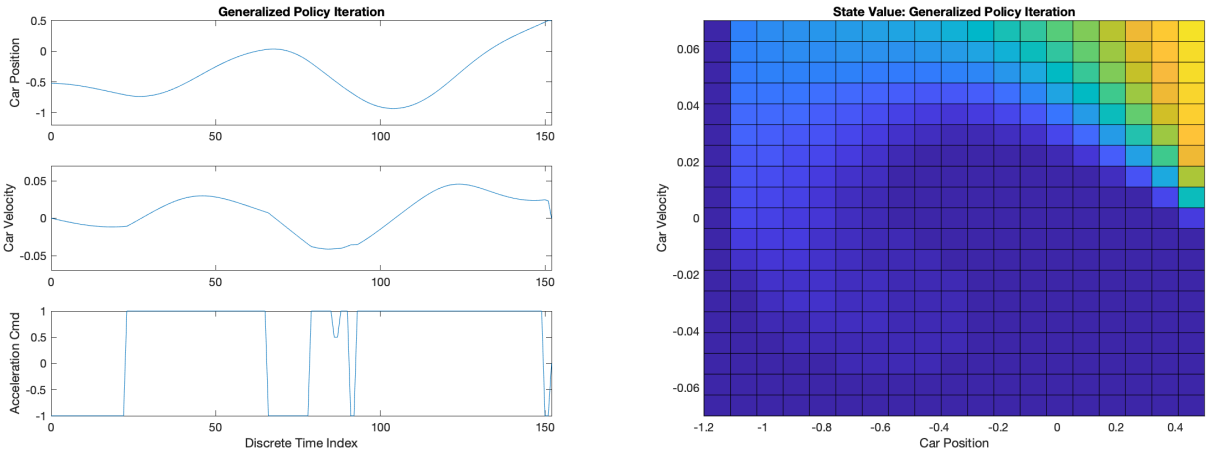

Figure 8: Mountain car trajectory (left) and the state value function heat-map (right)for running GPI on transition model discretized using the linear interpolation method.

When comparing the nearest neighbour and linear interpolation method, the latter appear to converge to a more optimal state-value function and produce a more optimal mountain car policy. Linear interpolation clearly produces a better representation of the continuous transition model.

2.2 Model Predictive Control

Model predictive controller (MPC) is another model-based approach to solving continuous state and action space control problem. The process involves converting the finite horizon control problem into a optimization problem. This optimization problem can be solved my non-linear solvers or through sequential quadratic programming (SQP) which involves linearizing the system model and recurrently solving a convex optimization problem. Equation 4 shows the control problem formulated as a non-linear optimization problem.

$$
\begin{aligned}
\min_{\mathbf{a}} \quad & (\mathbf{x}_N - \mathbf{x}_g)^T \mathbf{Q} (\mathbf{x}_N - \mathbf{x}_g) + \sum_{k=0} (\mathbf{x}_k - \mathbf{x}_g)^T \mathbf{Q} (\mathbf{x}_k - \mathbf{x}_g) + r a_k^2 \\
s.t. \quad & \mathbf{x}_{k+1}(1) = \mathbf{x}_k(1) + \mathbf{x}_k(2) + 0.001 a_k - 0.0025 cos(3\mathbf{x}_k(1)) && \forall k = 0, 1, ..., N-1 \\
& \mathbf{x}_{k+1}(2) = \mathbf{x}_k(2) + 0.001 a_k - 0.0025 cos(3\mathbf{x}_k(1)) && \forall k = 0, 1, ..., N-1 \\
& [-1.2, -0.07]^T \leq \mathbf{x}_k \leq [0.5, 0.07]^T && \forall k = 1, 2, ..., N \\
& -1 \leq a_k \leq 1 && \forall k = 0, 1, ..., N-1
\end{aligned}
\tag{4}
$$

Solving the non-linear MPC optimization is not feasible in real time. This the sequential quadratic programming is used. Figure 9 shows the resultant trajectory of the MPC with and without model noise. It is clear that MPC is capable of handling high system noise. As a matter of fact, it has been empirically verified that the MPC is able to handle any sane values of noise in the system.

Through some experiments it is concluded that the MPC has lower bound for the prediction horizon for which it a able to reach the goal. This lower bound is estimated to be close to 80 time-steps. An increase in the horizon does not a significant affect on the policy and thus the trajectory, though computation time increases significantly.

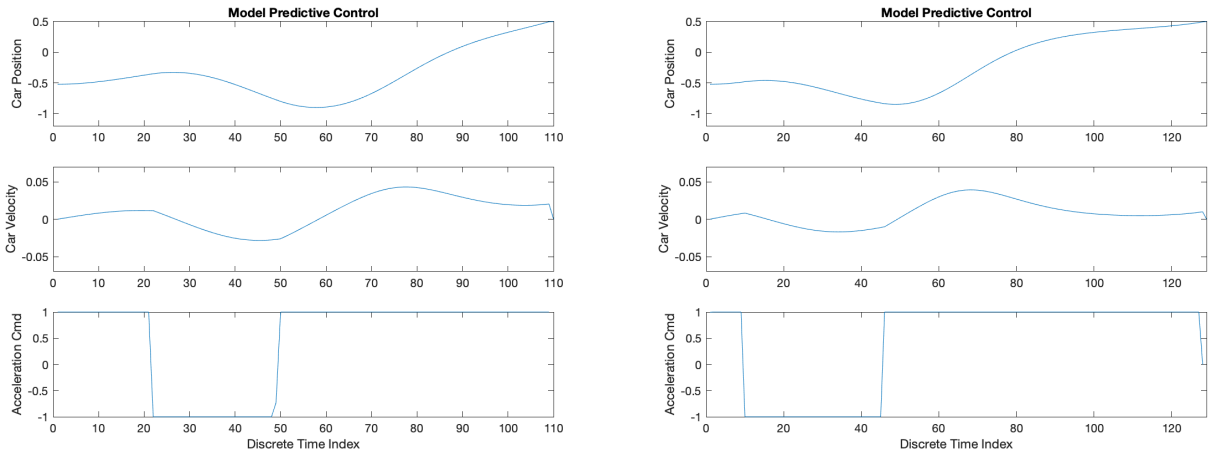

Figure 9: Mountain car trajectory (left) when running linear MPC with (left) and without (right) system noise (state standard deviation = $[3, 1]^T$).

References

[1] Andrew William Moore. Efficient memory-based learning for robot control. 1990.

APPENDIX A

GPI Code

```
function [V, policy_index] = generalized_policy_iteration(world,
    precision_pi, precision_pe, max_ite_pi, max_ite_pe)
    %% Initialization
    % MDP
    mdp = world.mdp;
    T = mdp.T; % transition_probability
    R = mdp.R; % Reward function
    gamma = mdp.gamma;

    % Dimensions
    num_actions = length(T);
    num_states = size(T{1}, 1);

    % Intialize value function
    V = zeros(num_states, 1);

    % Initialize policy
    % Note: Policy here encodes the action to be executed at state s. We
    %       use deterministic policy here (e.g., [0,1,0,0] means take
    %       action indexed 2)
    random_act_index = randi(num_actions, [num_states, 1]);
    policy = zeros(num_states, num_actions);
    for s = 1:1:num_states
        selected_action = random_act_index(s);
        policy(s, selected_action) = 1;
    end

    V_last_g = [];
    for  i = 1:max_ite_pe
        V_last_g = V;
```

```
%% [TODO] policy Evaluation (PE) (Section 2.6 of [1])
% V = ...;
V_last = [];

for j = 1:max_ite_pi
    V_last = V;
    Q = zeros(num_states, num_actions);
    for ai = 1:1:num_actions
        Q(:, ai) = diag(T{ai}*transpose(R{ai})) + gamma*T{ai}*V;
    end
    V = sum(policy.*Q, 2);
    V_error = norm(V-V_last);
    if V_error <= precision_pi
        disp(num2str(i) + ": " + num2str(j) +  ": Policy Evaluation
            Converged: " + num2str(V_error));
        break;
    end
end

%% [TODO] Policy Improvment (PI) (Section 2.7 of [1])
% policy = ...;
[temp, policy_index] = max(Q,[],2);
policy = zeros(num_states, num_actions);
for s = 1:1:num_states
    selected_action = policy_index(s);
    policy(s, selected_action) = 1;
end

final_error = norm(V-V_last_g);
% Check algorithm convergence
if final_error <= precision_pe
    disp(num2str(i) + ": Policy Iteration Converged: " + num2str(
```

```
                final_error));
            break;
        end
    end

    % Return deterministic policy for plotting
    [~, policy_index] = max(policy, [], 2);
end
```

LP Code

```
function [V, policy] = linear_programming(world)
    %% Initialization
    % MDP
    mdp = world.mdp;
    T = mdp.T; % transition_probability
    R = mdp.R; % Reward function
    gamma = mdp.gamma;

    % Dimensions
    num_actions = length(T);
    num_states = size(T{1}, 1);


    fprintf('\n\n\t########### Linear Programming ########')

    %% [TODO] Compute optimal value function (see [2] for reference)
    % V = ...;

    f = ones(num_states,1);
    A = [];
    b = [];
    for action_index = 1:1:num_actions
        A = [A; gamma*T{action_index} - eye(num_states)];
```

```
        b = [b; -diag(T{action_index}*R{action_index}')];
    end

    V = linprog(f, A, b);

    %% [TODO] Compute an optimal policy
    % policy = ...;
    Q = zeros(num_actions, num_states);
    for action_index = 1:1:num_actions
        Q(action_index, :) = diag(T{action_index}*R{action_index}') + gamma*
            T{action_index}*V;
    end

    [temp, policy] = max(Q,[],1);
end
```

Monte-Carlo Code

```
function [Q, policy_index] = ...
    monte_carlo(world, epsilon, k_epsilon, omega, training_iterations,
        episode_length)
    %% Initialization
    % MDP
    mdp = world.mdp;
    gamma = mdp.gamma;

    % States
    STATES = mdp.STATES;
    ACTIONS = mdp.ACTIONS;

    % Start and end State
    s_start = world.user_defn.s_start;
    s_goal = world.user_defn.s_goal;
    s_start_index = state_index_lookup(STATES, s_start);
```

```
    s_goal_index = state_index_lookup(STATES, s_goal);

    % Dimensionts
    num_states = size(STATES, 2);
    num_actions = size(ACTIONS, 2);

    % Create object for incremental plotting of reward after each episode
    windowSize = 10; %Sets the width of the sliding window fitler used in
        plotting
    plotter = RewardPlotter(windowSize);

    % Initialize Q
    Q = zeros(num_states, num_actions);

    % Initialize epsilon-soft policy
    random_act_index = randi(num_actions, [num_states, 1]);
    policy = zeros(num_states, num_actions);
    for s = 1:1:num_states
        selected_action = random_act_index(s);
        policy(s, selected_action) = 1;
    end

    %% On-policy Monte Carlo Algorithm (Section 2.9.3 of [1])
    for train_loop = 1:1:training_iterations
        R = [];
        episode_index = 0;
        cur_state_index = randi(num_states);
%         cur_state_index = s_start_index;
        %% Generate a training episode
%         while ( cur_state_index ~= s_start_index && ...
%                 cur_state_index ~= s_goal_index && ...
%                 episode_index < episode_length) || ...
%                 episode_index < 1
```

```
while episode_index < episode_length
    episode_index = episode_index + 1;
    % Sample current epsilon-soft policy
    [action, soft_policy] = sample_es_policy(...
                            policy(cur_state_index, :), epsilon);

    % Interaction with environment
    [next_state_index, ~, reward] = one_step_gw_model(world, ...
                                    cur_state_index, action, 1);

    % Log data for the episode
    R = [R; cur_state_index, action, reward];

    cur_state_index = next_state_index;
end

% Cummulated Discounted Reward
prev_sum = 0;
R_cumu = R(:,3);
for it = size(R,1):-1:1
    prev_sum = R_cumu(it) + gamma*prev_sum;
    R_cumu(it) = prev_sum;
end

% Update Q(s,a)
Used = zeros(num_states, num_actions);
for it = 1: size(R,1)
    if ~Used(R(it,1), R(it,2))
        Q(R(it,1), R(it,2)) = Q(R(it,1), R(it,2)) + ...
                    omega * (R_cumu(it) - Q(R(it,1), R(it,2)));
        Used(R(it,1), R(it,2)) = 1;
    end
end
```

```
        %% Update policy(s,a)
        [~, pi] = max(Q,[],2);
        policy = zeros(num_states, num_actions);
        for s = 1:num_states
            selected_action = pi(s);
            policy(s, selected_action) = 1;
        end

        %% Update the reward plot
        EpisodeTotalReturn = sum(R(:, 3)); % Sum of the reward obtained
           during the episode
        plotter = UpdatePlot(plotter, EpisodeTotalReturn);

        %% Decrease the exploration
        % Set k_epsilon = 1 to maintain constant exploration
        epsilon = epsilon * k_epsilon;
    end

    % Return deterministic policy for plotting
    [~, policy_index] = max(policy, [], 2);
    drawnow;
end

function [value, soft_policy] = sample_es_policy(policy, ep)

    size_u = length(policy);
    soft_policy = policy;
    cummu_dist = policy;

    for i = 1:size_u
        if policy(i) == 1
            soft_policy(i) = 1 - ep + ep/size_u;
```

```
        else
            soft_policy(i) = ep/size_u;
        end

        if i > 1
            cummu_dist(i) = soft_policy(i) + cummu_dist(i-1);
        end
    end

    rand_n = rand([1,1],'double');
%     disp("Policy: " + num2str(soft_policy) + " Rand: " + num2str(rand_n));

    for i = 1:size_u
        if rand_n <= cummu_dist(i)
            value = i;
            return
        end
    end
    value = size_u;
end
```

Q-Learning Code

```
function [Q, policy_index] = q_learning(world, epsilon, k_epsilon, omega,
    training_iterations, episode_length, noise_alpha)
    %% Initialization
    % MDP
    mdp = world.mdp;
    gamma = mdp.gamma;

    % States
    STATES = mdp.STATES;
    ACTIONS = mdp.ACTIONS;
```

```
% Dimensionts
num_states = size(STATES, 2);
num_actions = size(ACTIONS, 2);

% Create object for incremental plotting of reward after each episode
windowSize = 10; %Sets the width of the sliding window fitler used in
    plotting
plotter = RewardPlotter(windowSize);

% Initialize Q
Q = zeros(num_states, num_actions);

% Initialize epsilon-soft policy
random_act_index = randi(num_actions, [num_states, 1]);
policy = zeros(num_states, num_actions);
for s = 1:1:num_states
    selected_action = random_act_index(s);
    policy(s, selected_action) = 1;
end

%% Q-Learning Algorthim (Section 2.9 of [1])
for train_loop = 1:1:training_iterations
    %% [TODO] Generate a training episode
    R = [];
    episode_index = 0;
    cur_state_index = randi(num_states);
    while episode_index < episode_length
        episode_index = episode_index + 1;
        % Sample current epsilon-soft policy
        [action, soft_policy] = sample_es_policy(...
                                policy(cur_state_index, :), epsilon);

        % Interaction with environment% Note: 'next_state_noisy_index'
```

```
        below simulates state
    %       observarions corrupted with noise. Use this for
    %       Q-learning correspondingly for the last part of
    %       Problem 2.2 (d)
    % [next_state_index, next_state_noisy_index, reward] = ...
    %   one_step_gw_model(world, cur_state_index, action,
        noise_alpha);
    [next_state_index, ~, reward] = one_step_gw_model(world, ...
                                    cur_state_index, action,
                                        noise_alpha);

    % Log data for the episode
    R = [R; cur_state_index, action, reward];

    % Update Q(s,a)
    Q(cur_state_index, action) = Q(cur_state_index, action) +...
      omega * (reward + gamma * max(Q(next_state_index, :)) -...
                Q(cur_state_index, action));

    cur_state_index = next_state_index;
end

%% [TODO] Update policy(s,a)
[~, pi] = max(Q,[],2);
policy = zeros(num_states, num_actions);
for s = 1:num_states
    selected_action = pi(s);
    policy(s, selected_action) = 1;
end

%% [TODO] Update the reward plot
EpisodeTotalReturn = sum(R(:, 3)); %Sum of the reward obtained
    during the episode
```

```
        plotter = UpdatePlot(plotter, EpisodeTotalReturn);
%           drawnow;
%           pause(0.1);

        %% Decrease the exploration
        %Set k_epsilon = 1 to maintain constant exploration
        epsilon = epsilon * k_epsilon;

    end

    % Return deterministic policy for plotting
    [~, policy_index] = max(policy, [], 2);
end

function [value, soft_policy] = sample_es_policy(policy, ep)

    size_u = length(policy);
    soft_policy = policy;
    cummu_dist = policy;

    for i = 1:size_u
        if policy(i) == 1
            soft_policy(i) = 1 - ep + ep/size_u;
        else
            soft_policy(i) = ep/size_u;
        end

        if i > 1
            cummu_dist(i) = soft_policy(i) + cummu_dist(i-1);
        end
    end

    rand_n = rand([1,1],'double');
```

```
%     disp("Policy: " + num2str(soft_policy) + " Rand: " + num2str(rand_n));

    for i = 1:size_u
        if rand_n <= cummu_dist(i)
            value = i;
            return
        end
    end
    value = size_u;
end
```

Nearest Neighbour Code

```
function [T, R] = build_stochastic_mdp_nn(world, T, R, num_samples)
    % Extract states and actions
    STATES = world.mdp.STATES;
    ACTIONS = world.mdp.ACTIONS;

    % Dimensions
    num_states = size(STATES, 2);
    num_actions = size(ACTIONS, 2);

    % Noise Parameters
    var_p = 0.14/40;
%     var_p = 0.0;
    std_state = [var_p, var_p];

    % Value Bounds
    pos_bounds = world.param.pos_bounds;
    vel_bounds = world.param.vel_bounds;

    % Goal
    s_goal = world.param.s_goal;
    s_goal_indices = [];
```

```
for it = 1:size(s_goal,2)
    s_goal_indices = [s_goal_indices, ...
        nearest_state_index_lookup(STATES, s_goal(:, it))];
end

% Misc
visits = {};
for action_index = 1:1:num_actions
    visits{action_index} = zeros(num_states, num_states);
end

% Loop through all possible states
for state_index = 1:1:num_states
    cur_state = STATES(:, state_index);
    x = cur_state(1);
    v = cur_state(2);
    fprintf('building model... state %d\n', state_index);

    % Apply each possible action
    for action_index = 1:1:num_actions
        action = ACTIONS(:, action_index);

        % Build a stochastic MDP based on Nearest Neighbour
        % Note: The function 'nearest_state_index_lookup' can be used
        % to find the nearest node to a countinuous state
        for samples = 1:1:num_samples

            % Get next state index
            [~,next_state,reward,~] = one_step_mc_model_noisy(world,
                cur_state, action, std_state);
            next_state_index = nearest_state_index_lookup(STATES,
                next_state);
```

```
                % Update transition and reward models
                T{action_index}(state_index, next_state_index) = 1/
                    num_samples + ...
                     T{action_index}(state_index, next_state_index);

                R{action_index}(state_index, next_state_index) = reward +
                    ...
                     R{action_index}(state_index, next_state_index);
                visits{action_index}(state_index, next_state_index) = 1 +
                    ...
                     visits{action_index}(state_index, next_state_index);

            end
        end
    end
    for action_index = 1:1:num_actions
        non_zero = find(R{action_index} ~= 0.0);
        R{action_index}(non_zero) = R{action_index}(non_zero)./...
            visits{action_index}(non_zero);
    end
end
```

Linear Interpolation Code

```
function [T, R] = build_stochastic_mdp_li(world, T, R)
        % Extract states and actions
    STATES = world.mdp.STATES;
    ACTIONS = world.mdp.ACTIONS;

    % Number of discrete states and actions
    num_states = size(STATES, 2);
    num_actions = size(ACTIONS, 2);

    % State space dimension
```

```
dim_state = size(STATES, 1);

    % Unique values
for i = 1:1:dim_state
    unique_states{i} = unique(STATES(i,:));
end

% Loop through all possible states
for state_index = 1:1:num_states
    cur_state = STATES(:, state_index);
    fprintf('building model... state %d\n', state_index);

    % Apply each possible action
    for action_index = 1:1:num_actions
        action = ACTIONS(:, action_index);

        % Propagate forward
        [next_state, reward, ~] = world.one_step_model(world, ...
            cur_state, action);

        % Find four vertices enclosing next state index
        for i = 1:1:dim_state
            % find cloest discretized values along state dimension i
            node_index_temp = knnsearch(unique_states{i}', next_state(i)
                , 'K', 2);
            node_value_temp = unique_states{i}(node_index_temp);

            % for each state dimension i, store the min-max bounds
            box_min = min(node_value_temp);
            box_max = max(node_value_temp);
            node_value(i,1:2) = [box_min, box_max];

            % normalize next state values
```

```
                next_state_normalized(i,1) = ...
                    (next_state(i,1) - box_min) / (box_max - box_min);
            end

            % node values (for two-dim state space)
            node(1:2,1) = [node_value(1,1); node_value(2,1)]; % lower-left
            node(1:2,2) = [node_value(1,2); node_value(2,1)]; % lower-right
            node(1:2,3) = [node_value(1,2); node_value(2,2)]; % upper-right
            node(1:2,4) = [node_value(1,1); node_value(2,2)]; % upper-left

            % [TODO] Assign probability to adjacent nodes (bilinear)
            x = next_state_normalized(1);
            y = next_state_normalized(2);
            prob(1) = (1-x)*(1-y); % min min
            prob(2) = (x)*(1-y); % max min
            prob(3) = (x)*(y); % max max
            prob(4) = (1-x)*(y); % min max

            % Update probability and reward for each node
            for i = 1:1:4
                node_index = nearest_state_index_lookup(STATES, node(:,i));

                % Update transition and reward models
                T{action_index}(state_index, node_index) = prob(i);
                R{action_index}(state_index, node_index) = reward;
            end

        end
    end
end
```

MPC Code

```
clear all;
```

```
close all;
clc;

%% General
% Add path
addpath(genpath(pwd));

% MPC parameters
n_lookahead = 100; % MPC prediction horizon
n_mpc_update = 1; % MPC update frequency
use_model = false;
add_noise = true;

% Cost function parameters
Q = diag([100, 0]); % not penalizing velocity
r = 0;

% Initial state
cur_state = [-pi/6; 0]; % [-pi/6; 0];
goal_state = [0.5; 0.05];
state_stack = cur_state;
input_stack = [];

% State and action bounds
pos_bounds = [-1.2, 0.5]; % state 1: position
vel_bounds = [-0.07, 0.07]; % state 2: velocity
acc_bounds = [-1, 1]; % action: acceleration

% Plotting parameters
linecolor = [1, 1, 1].*0.5;
fontcolor = [1, 1, 1].*0.5;
fontsize = 12;
```

```
% Max number of time steps to simulate
max_steps = 500;

% Standard deviation of simulated Gaussian measurement noise
noise = [3; 1];

% Result and plot directory
save_dir = './results/';
mkdir(save_dir);

%% Problem 2.2: (d) Solving mountain car problem with MPC
% State and action bounds
state_bound = [pos_bounds; vel_bounds];
action_bound = [acc_bounds];

% Struct used in simulation and visualization scripts
world.param.pos_bounds = pos_bounds;
world.param.vel_bounds = vel_bounds;
world.param.acc_bounds = acc_bounds;

% Action and state dimensions
dim_state = size(state_bound, 1);
dim_action = size(action_bound, 1);

% MPC implmentation
tic;
for k = 1:1:max_steps

    if mod(k, n_mpc_update) == 1 || n_mpc_update == 1
        fprintf('updating inputs...\n');
        % Get cost Hessian matrix
        S = get_cost(r, Q, n_lookahead);
```

```
% Lower and upper bounds
lb = [repmat(action_bound(1),n_lookahead,1); ...
     repmat(state_bound(:,1),n_lookahead,1)];
ub = [repmat(action_bound(2),n_lookahead,1); ...
     repmat(state_bound(:,2)+[0;0],n_lookahead,1)];

% Optimize state and action over prediction horizon
if k <= 1
    % Solve nonlinear MPC at the first step
    if k == 1
        initial_guess = randn(n_lookahead*(dim_state+dim_action), 1)
            ;
    else
        initial_guess = x;
    end

    % Cost function
    sub_states = [repmat(0,n_lookahead,1); ...
        repmat(goal_state, n_lookahead,1)];
    fun = @(x) (x - sub_states)'*S*(x - sub_states);

    % Temporary variables used in 'dyncons'
    save('params', 'n_lookahead', 'dim_state', 'dim_action');
    save('cur_state', 'cur_state');

    % Solve nonlinear MPC
    % x is a vector containing the inputs and states over the
    % horizon [input,..., input, state', ..., state']^T
    options = optimoptions(@fmincon, 'MaxFunctionEvaluations', ...
        1e5, 'MaxIterations', 1e5, 'Display', 'iter');
    [x,fval] = fmincon(fun, initial_guess, [], [], [], [], ...
        lb, ub, @dyncons, options);
else
```

```
% ================== [TODO] QP Implementation ================
% Problem 2.2: (d) Quadratic Program optimizing state and
% action over prediction horizon

% Feedback state used in MPC updates
% 'cur_state' or 'cur_state_noisy'
if ~add_noise
    cur_state_mpc_update = cur_state;
else
    cur_state_mpc_update = cur_state_noisy;
end

% Solve QP (e.g., using Matlab's quadprog function)
% Note 1: x is a vector containing the inputs and states over
%         the horizon [input,..., input, state', ..., state']^T
% Note 2: The function 'get_lin_matrices' computes the
%         Jacobians (A, B) evaluated at an operation point
x_prev = cur_state_mpc_update;
a_curr = [];
states_curr = [];
Aeq = zeros(n_lookahead*(dim_state), n_lookahead*(dim_state+
   dim_action));
Aeq(:, n_lookahead*(dim_action)+1:end) = ...
                        - eye(n_lookahead*(dim_state));
% Populate Aeq
for s = 1:n_lookahead
    % Get linearization points
    actions = n_lookahead*(dim_action);
    if s < n_lookahead
        a_bar = cur_mpc_inputs(:, s);
        if ~use_model
            x_bar = cur_mpc_states(:, s);
        end
```

```
    else
        a_bar = 0;
        if ~use_model
            [x_bar, ~, ~] = ...
            one_step_mc_model(world, x_prev, a_bar);
        end
    end
    if use_model
        [x_bar, ~, ~] = ...
            one_step_mc_model(world, x_prev, a_bar);
    end
    a_curr = [a_curr; a_bar];
    states_curr = [states_curr; x_bar];
    x_prev = x_bar;

    % Get linearized model for current state and action
    [A_, b_] = get_lin_matrices(x_bar, a_bar);

    % Populate Aeq
    if s > 1
        mat = A_ * Aeq((s-2)*dim_state+1:(s-1)*dim_state, 1:(s
            -1)*dim_action );
        Aeq((s-1)*dim_state+1:(s)*dim_state, 1:(s-1)*dim_action)
            = mat;
    end
    Aeq((s-1)*dim_state+1:(s)*dim_state, (s)*dim_action) = b_;
end

x = [a_curr; states_curr];

beq = Aeq * x;
f = -(S + S')*sub_states;
H = S;
```

```
        x = quadprog(H,f,[],[],Aeq, beq, lb, ub);
        % ============================================================
    end

    % Separate inputs and states from the optimization variable x
    inputs = x(1:n_lookahead*dim_action);
    states_crossterms = x(n_lookahead*dim_action+1:end);
    position_indeces = 1:2:2*n_lookahead;
    velocity_indeces = position_indeces + 1;
    positions = states_crossterms(position_indeces);
    velocities = states_crossterms(velocity_indeces);

    % Variables if not running optimization at each time step
    cur_mpc_inputs = inputs';
    cur_mpc_states = [positions'; velocities'];
end

% Propagate
action = cur_mpc_inputs(1);
[cur_state, cur_state_noisy, ~, is_goal_state] = ...
    one_step_mc_model_noisy(world, cur_state, action, noise);

% Remove first input
cur_mpc_inputs(1) = [];
cur_mpc_states(:,1) = [];

% Save state and input
state_stack = [state_stack, cur_state];
input_stack = [input_stack, action];

% Plot
grey = [0.5, 0.5, 0.5];
```

```
hdl = figure(1);
hdl.Position(3) = 1155;
clf;
subplot(3,2,1);
    plot(state_stack(1,:), 'linewidth', 3); hold on;
plot(k+1:k+length(cur_mpc_states(1,:)), cur_mpc_states(1,:), 'color',
   grey);
ylabel('Car Position');
subplot(3,2,3);
plot(state_stack(2,:), 'linewidth', 3); hold on;
plot(k+1:k+length(cur_mpc_states(2,:)), cur_mpc_states(2,:), 'color',
   grey);
ylabel('Car Velocity');
subplot(3,2,5);
plot(input_stack(1,:), 'linewidth', 3); hold on;
plot(k:k+length(cur_mpc_inputs)-1, cur_mpc_inputs, 'color', grey);
xlabel('Discrete Time Index');
ylabel('Acceleration Cmd');
subplot(3,2,[2,4,6]);
    xvals = linspace(world.param.pos_bounds(1), world.param.pos_bounds
       (2));
    yvals = get_car_height(xvals);
    plot(xvals, yvals, 'color', linecolor, 'linewidth', 1.5); hold on;
plot(cur_state(1), get_car_height(cur_state(1)), 'ro', 'linewidth', 2);
xlabel('x Position');
ylabel('y Position');
pause(0.1);

% Break if goal reached
if is_goal_state
    fprintf('goal reached\n');
    break
end
```

```
end
compute_time = toc;

% Visualization
plot_visualize = true;
plot_title = 'Model Predictive Control';
hdl = visualize_mc_solution_mpc(world, state_stack, input_stack, ...
    plot_visualize, plot_title, save_dir);

% Save results
save(strcat(save_dir, 'mpc_results.mat'), 'state_stack', 'input_stack');
```