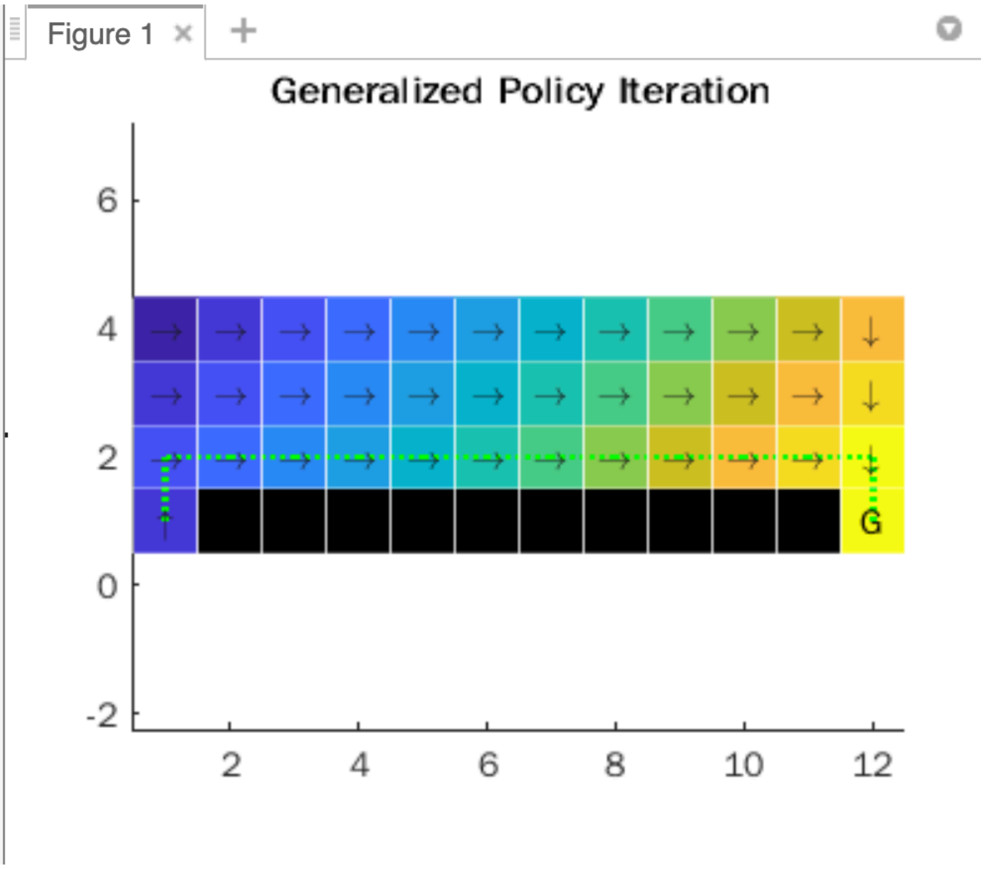
4.1)

a) generalized_policy_iteration.m

```matlab
% generalized_policy_iteration: Function solving the given MDP using the
%                               Generalized Policy Iteration algorithm
%
% Inputs:
%       world:                  A structure defining the MDP to be solved
%       precision_pi:           Maximum value function change before
%                               terminating Policy Improvement step
%       max_ite_pi:             Maximum number of iterations for Policy
%                               Improvement loop
%       precision_pe:           Maximum value function change before
%                               terminating Policy Evaluation step
%       max_ite_pe:             Maximum number of iterations for Policy
%                               Evaluation loop
%
% Outputs:
%       V:                      An array containing the value at each state
%       policy_index:           An array summarizing the index of the
%                               optimal action index at each state
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.03.07, SZ]    first version

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;
    R = mdp.R;
    gamma = mdp.gamma;
    iteration_pi = 0;
    % 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

    while true
        %% [TODO] policy Evaluation (PE) (Section 2.6 of [1])
       iterations_pe = 0
       iteration_pi = iteration_pi + 1
       while iterations_pe <= max_ite_pe
        delta = 0
        iterations_pe = iterations_pe + 1
        for s = 1:1:num_states %loop for each state

            v = V(s,1); %initialize v value
            cur_state_index = s
            action_index = find(policy(s,:))
            noise_alpha = 0
            [next_state_index, next_state_noisy_index, reward] = ...
            one_step_gw_model(world, cur_state_index, action_index, noise_alpha)
%compute next state, reward of transition when applying a = pi(s)
            V(s,1) = reward + gamma*V(next_state_index,1) %compute value of
policy

            abs_diff = abs(v-V(s,1));
            delta = max(abs_diff,delta);
        end

        if delta < precision_pe

        break

        end

       end

        % V = ...;

        %% [TODO] Policy Improvment (PI) (Section 2.7 of [1])
        policyISstable = true;
        for s = 1:1:num_states

            b = policy(s,:);

            %compute argmax of cumulative reward function:
            cur_state_index = s;
            noise_alpha = 0;

            for a = 1:4
            [next_state_index, next_state_noisy_index, reward] =
one_step_gw_model(world, cur_state_index, a, noise_alpha)
            V_temp = reward + gamma*V(next_state_index,1);

            if a == 1
            temp = V_temp
            end

            if V_temp >= temp

                temp = V_temp;
                argmax_a = a;   %computing which action maximizes cumulative
reward

            end

            end

            policy(s,:) = zeros(1,4);   %updating policy
            policy(s,argmax_a) = 1;

         if find(b) ~= find(policy(s,:))
            fprintf('policy not stable')
            policyISstable = false;
         end

        end

        if policyISstable == true

            break
        end
        % policy = ...;

        % Check algorithm convergence
        % if ...
        %       break
        % end
    end
    iteration_pi
    % Return deterministic policy for plotting
    [~, policy_index] = max(policy, [], 2);
end
```

Heat Map and Results:



Difference between VI and PI:

In value iteration, the policy is evaluated only once and then improved. In policy iteration, the policy evaluation step is terminated only when the termination condition is satisfied. That is, an accurate estimate of the value function of a particular policy is computed first and then the policy is improved based on the accurate value function estimate.

Value iteration converges faster than PI. As the policy evaluation step takes place only once in value iteration, VI converges to optimal policy faster.

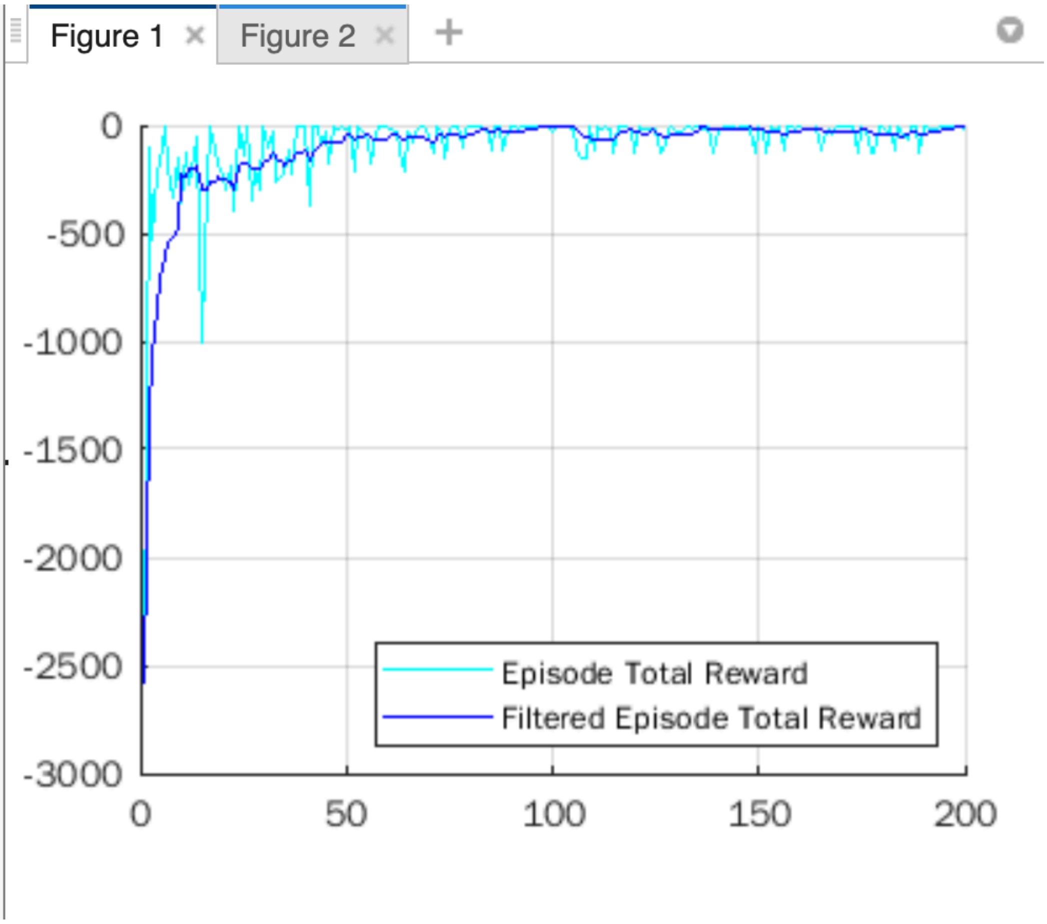
c) Monte Carlo

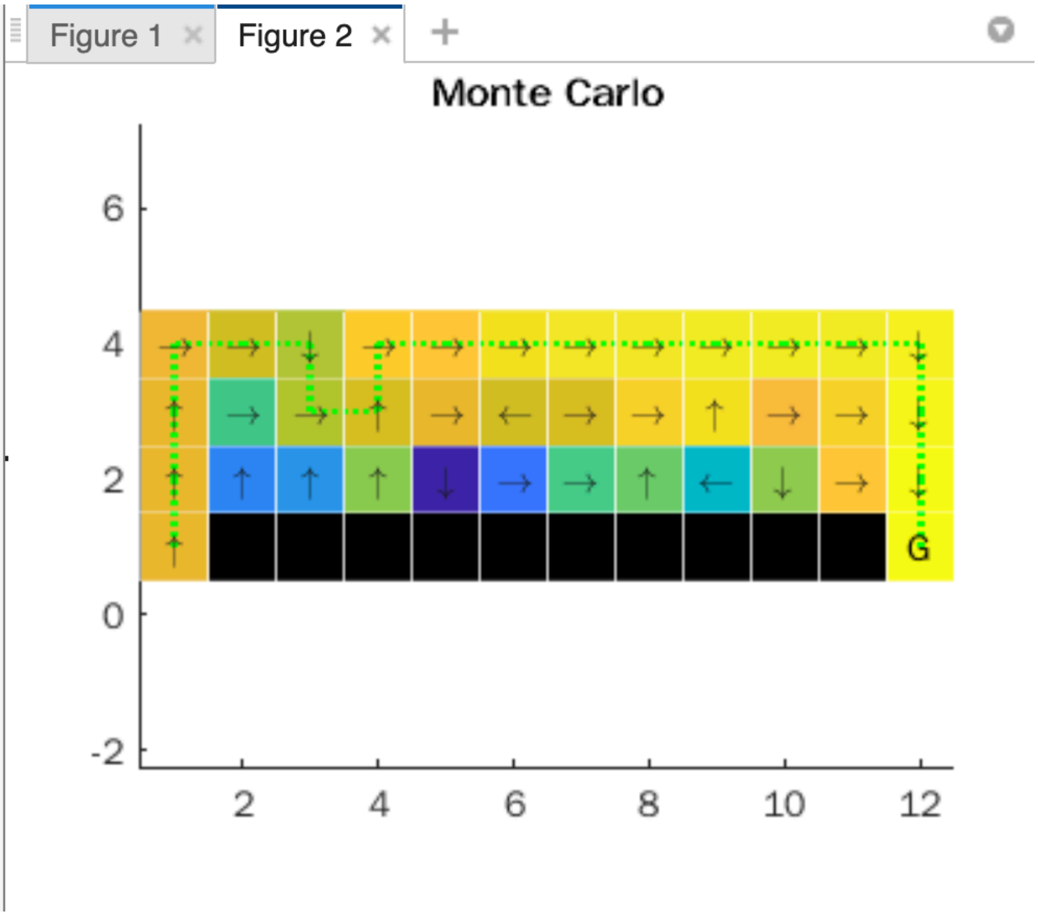
1) code: monte_carlo.m:

```
% monte_carlo: Function solving the given MDP using the on-policy Monte
%              Carlo method
%
% Inputs:
%      world:                A structure defining the MDP to be solved
%      epsilon:              A parameter defining the 'sofeness' of the
%                            epsilon-soft policy
%      k_epsilon:            The decay factor of epsilon per iteration
%      omega:                Learning rate for updating Q
%       training_iterations:    Maximum number of training episodes
%       episode_length:         Maximum number of steps in each training
%                               episodes
%
% Outputs:
%       Q:                      An array containing the action value for
%                               each state-action pair
%       policy_index:           An array summarizing the index of the
%                               optimal action index at each state
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.03.07, SZ]    first version

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;

    % 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);
    terminal_state = 12
    % [TODO] Initialize epsilon-soft policy
    % policy = ...; % size: num_states x num_actions
    policy = zeros(num_states, num_actions);
    policy = initialize_random_policy(epsilon,num_states,num_actions);
    %% On-policy Monte Carlo Algorithm (Section 2.9.3 of [1])
    initial_state = randi([1, num_states]);
    curr_state = initial_state;
    cur_state_index = curr_state;
    state_sequence = [curr_state];
    reward_sequence = [];
    action_sequence = [];

    for train_loop = 1:1:training_iterations
        %% [TODO] Generate a training episode
        initial_state = randi([1, num_states]); %randomly initialize states
        cur_state_index = initial_state;
        R = 0; %Initialize Return
        state_sequence = [curr_state];
        reward_sequence = []; %initialize reward sequence
        action_sequence = []; %initialize state  sequence
        episode_index = 0
         while cur_state_index ~= terminal_state & episode_index <
episode_length % episode termination criteria
            episode_index = episode_index + 1;
            policy_prob = policy(cur_state_index,:)
            % Sample current epsilon-soft policy
            action =sample_from_epsilon_policy(epsilon,policy_prob);

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


            cur_state_index = next_state_index;
            % Log data for the episode
            % ...
        end
        N = length(state_sequence);
        i = 0;
        reward_sequence
        action_sequence
        state_sequence
        for i = 1:N-1

            s = state_sequence(N-i);
            a = action_sequence(N-i);
            r = reward_sequence(N-i)
            R = gamma*R + r ; % cumulative return
            Q(s,a) = Q(s,a) + omega*(R - Q(s,a));


        end
        R
        % Update Q(s,a)
        % Q = ...;

        %% [TODO] Update policy(s,a)
         for i = 1:N-1

             x = state_sequence(N-i);
             u_optim = arg_max_Q(Q,x);

          for a = 1:4

            if a == u_optim
```

```
             policy(x,u_optim) = 1 - 0.75*epsilon;
            else
             policy(x,a) = 0.25*epsilon;
            end

          end

         end

        % policy = ...;

        %% [TODO] Update the reward plot
        EpisodeTotalReturn = R; % 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
```

Results (Monte Carlo) (epsilon = 0.2)



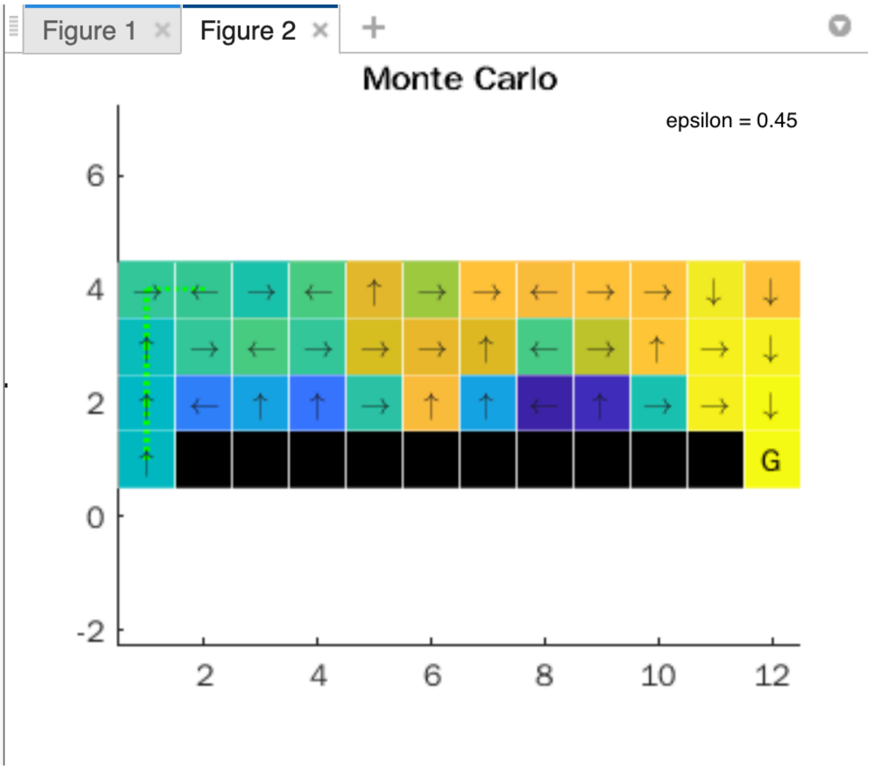


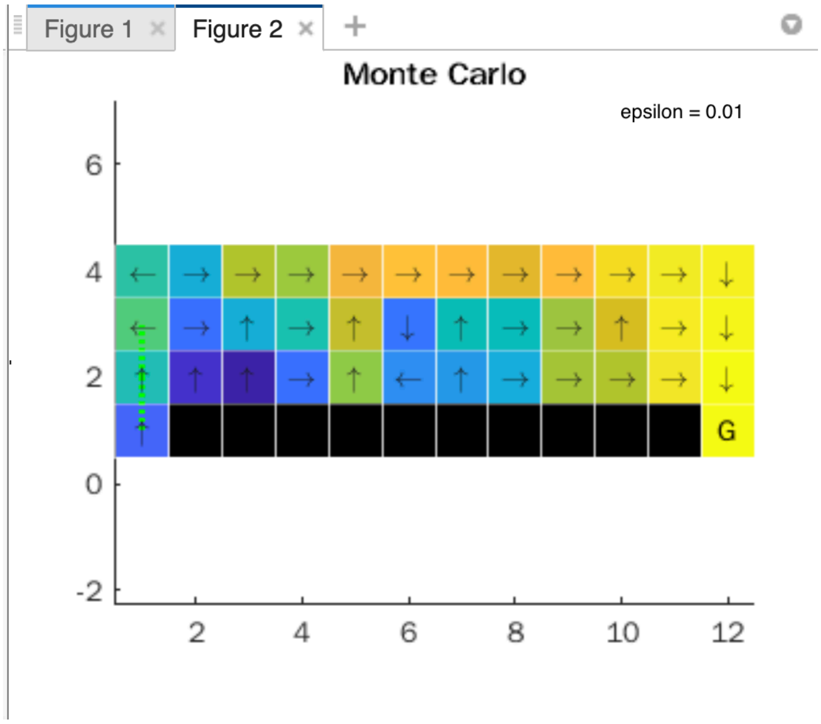
The algorithm computes a suboptimal policy. Even after multiple runs, the algorithm could not compute the optimal greedy policy(parameters: iterations = 500, episode_length = 500). Furthermore, in some cases the algorithm could not compute a path between the start point and the goal point.

2. Impact of varying policy parameter epsilon:

The soft policy parameter epsilon controls the frequency of exploratory actions performed by the agent.

Results for different values of epsilon are shown below:





For small values of epsilon, it is observed that computed policy is close to optimal policy for states nearby the goal states. However, for small values of epsilon, the agent might get stuck at a local maxima as the agent does not take too many exploratory actions.

For large values of epsilon, it is observed that the agent takes too many exploratory actions. Because of this, the computed policy is non optimal. In this case, more iterations would be required to compute optimal policies.

From the experiments it is clear that the epsilon parameter must be carefully tuned, depending upon the problem setup. A large value might result in too many exploratory moves while a smaller value would make the agent stuck at a local maxima.

d)

Code: q_learning.m:

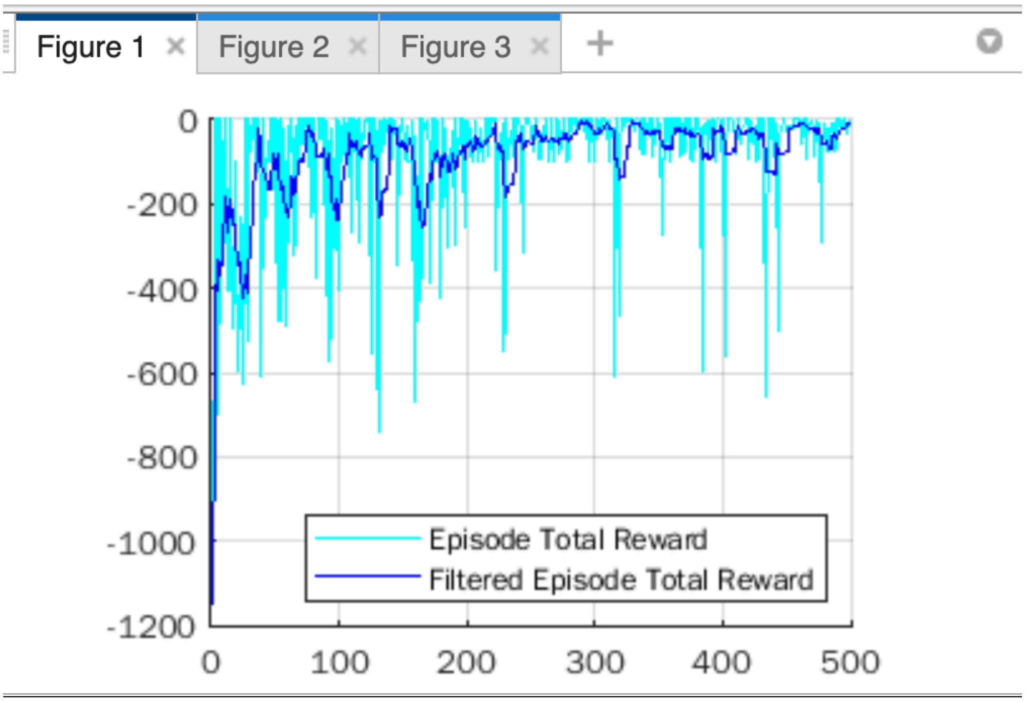
```
% q_learning: Function solving the given MDP using the off-policy
%             Q-Learning method
%
% Inputs:
%       world:                  A structure defining the MDP to be solved
%       epsilon:                A parameter defining the 'sofeness' of the
%                               epsilon-soft policy
%       k_epsilon:              The decay factor of epsilon per iteration
%       omega:                  Learning rate for updating Q
%       training_iterations:    Maximum number of training episodes
%       episode_length:         Maximum number of steps in each training
%                               episodes
%       noise_alpha:            A parameter that controls the noisiness of
%                               observation (observation is noise-free when
%                               noise_alpha is set to 1 and is more
%                               corrupted when it is set to values closer
%                               to 0)
%
% Outputs:
```

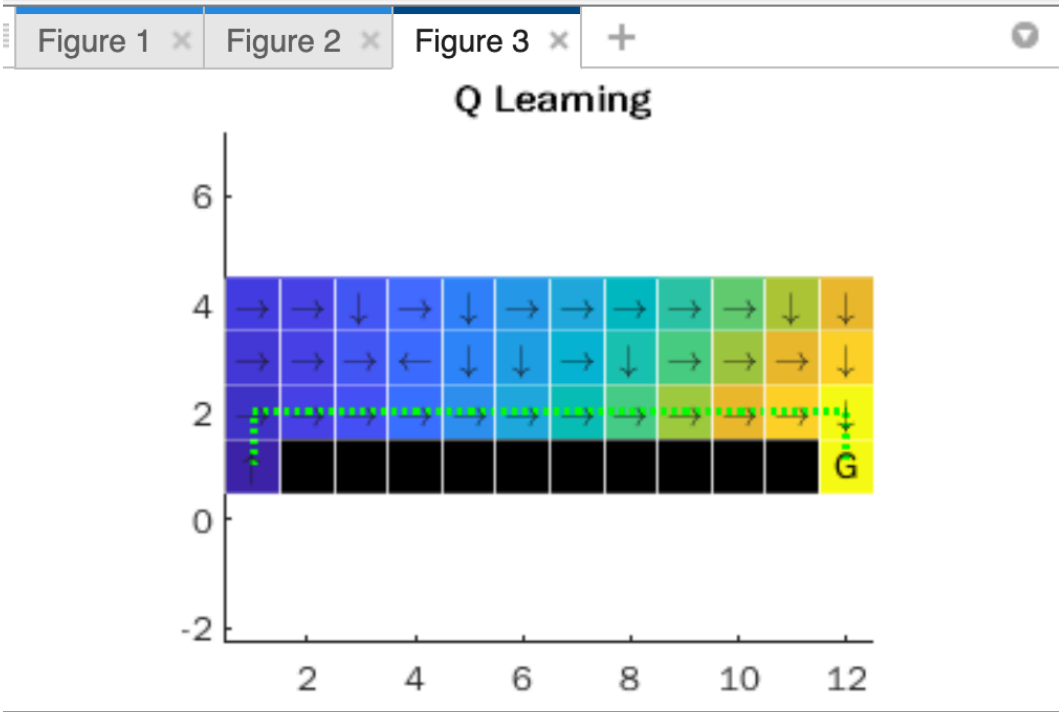
```
%       Q:                      An array containing the action value for
%                               each state-action pair
%       policy_index:           An array summarizing the index of the
%                               optimal action index at each state
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.03.07, SZ]    first version

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;
    terminal_state = 12
    % 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);

    % [TODO] Initialize epsilon-soft policy
    % policy = ...; % size: num_states x num_actions
    policy = initialize_random_policy(epsilon,num_states,num_actions);
    %% Q-Learning Algorthim (Section 2.9 of [1])
    for train_loop = 1:1:training_iterations
        %% [TODO] Generate a training episode
        initial_state = randi([1, num_states]); %randomly initialize states
%         initial_state = mdp.s_start_index;
```

```
        cur_state_index = initial_state;
        reward_sequence = [];
        episode_index = 0;
        R = 0;
        while cur_state_index ~= terminal_state & episode_index <=
episode_length
            % Sample current epsilon-soft policy
            policy_prob = policy(cur_state_index,:);
            % Sample current epsilon-soft policy
            action =sample_from_epsilon_policy(epsilon,policy_prob);
            episode_index = episode_index + 1;
            % 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);
            %Q learning update rule::
            argmax_u_prime = arg_max_Q(Q, next_state_index);
            Q(cur_state_index,action) = Q(cur_state_index,action) +
omega*(reward + gamma*Q(next_state_index,argmax_u_prime) -
Q(cur_state_index,action));
            reward_sequence = [reward_sequence, reward];

            % Log data for the episode
            % ...
            x = cur_state_index;
            u_optim = arg_max_Q(Q,x);

          for a = 1:4

            if a == u_optim
             policy(x,u_optim) = 1 - 0.75*epsilon;
            else
             policy(x,a) = 0.25*epsilon;
            end


            % Update Q(s,a)
            % Q = ...;
         end

        %% [TODO] Update policy(s,a)
        % policy = ...;
           N = length(reward_sequence);
          for i = 1:N-1

            r = reward_sequence(N-i+1);
            R = gamma*R + r ; % cumulative return

          end

          cur_state_index = next_state_index;
        end

        %% [TODO] Update the reward plot
        EpisodeTotalReturn = R %Sum of the reward obtained during the episode
        plotter = UpdatePlot(plotter, EpisodeTotalReturn);
        drawnow;
        pause(0.1);

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

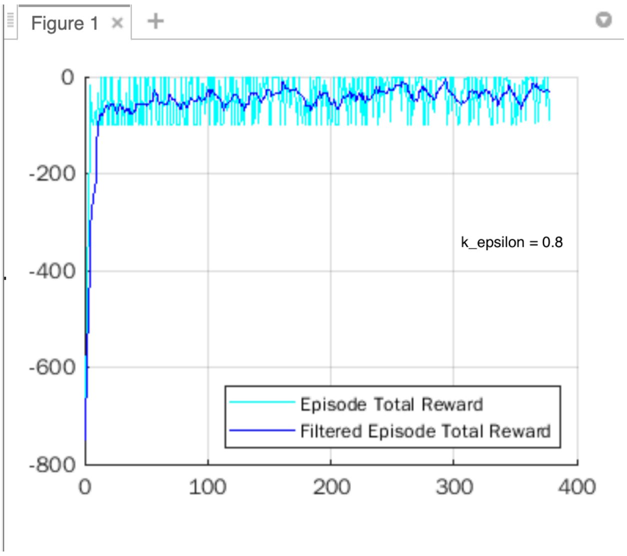
```
    end

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

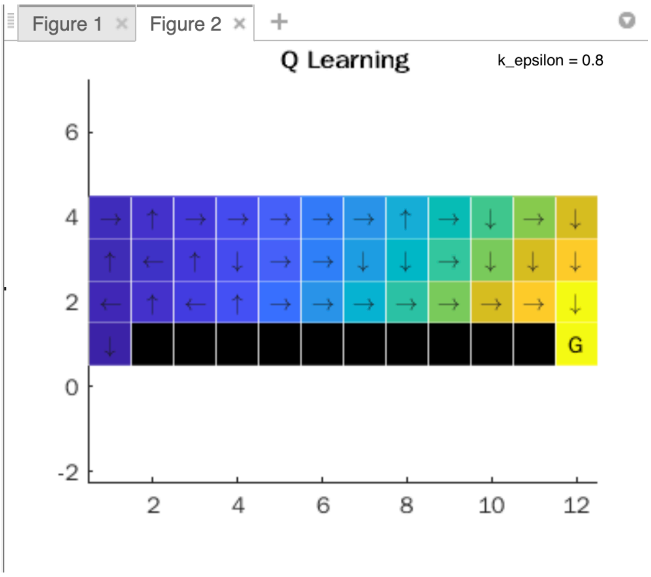
Simulation Results:





Impact of changing decay parameter k_epsilon





→	↑	→	→	→	→	→	↑	→	↓	→	↓
↑	←	↑	↓	→	→	↓	↓	→	↓	↓	↓
←	↑	←	↑	→	→	→	→	→	→	→	↓
↓											G

When k_epsilon is decreased from 1, exploration is reduced. This means that the agent would not take non greedy actions frequently and hence the Q function does not converge to optimal Q function. Because of this, the algorithm couldn’t compute a path from the start point to the goal location.

Difference between monte carlo and q learning algorithm:

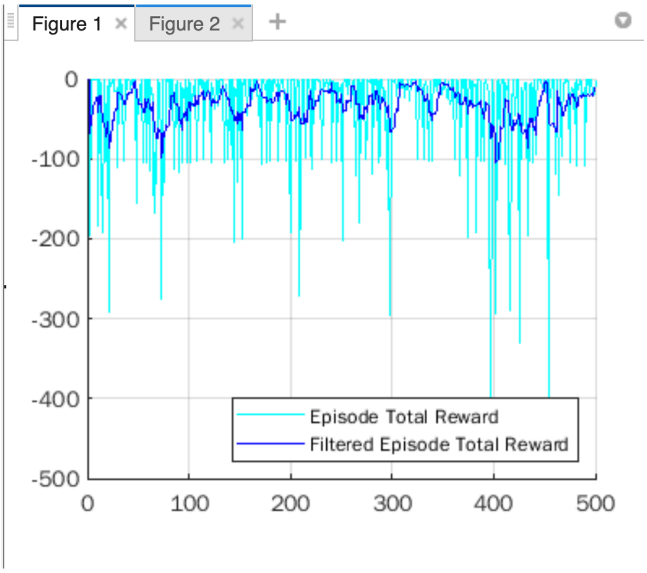
Monte Carlo is an on policy method while q learning is off policy.

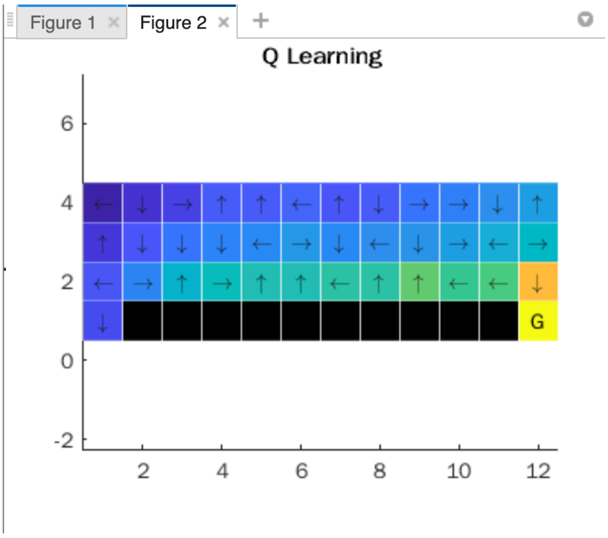
It is observed that the monte carlo algorithm computes suboptimal policies, while the policy from q learning algorithm is much closer to the optimal policy. Also, it is observed that the cumulative reward quickly converges to higher values for q learning in less number of iterations. Therefore, q learning algorithm is computationally more efficient than monte carlo.

Q learning with noise:

noise_alpha = 0.5

Results:





←	↓	→	↑	↑	←	↑	↓	→	→	↓	↑
↑	↓	↓	↓	←	→	↓	←	↓	→	←	→
←	→	↑	→	↑	↑	←	↑	↑	←	←	↓
↓	■	■	■	■	■	■	■	■	■	■	G

The computed policy is non optimal, when noise is added to the measurements (noise_alpha = 0.5). When noise_alpha was set to 1 (minimum noise), the algorithm was able to compute near optimal policies.

4.2)

4.2.a)

1.1 Code - build_stochastic_mdp_nn.m

```
% build_stochastic_mdp_nn: Function implementing the Nearest Neighbour
%                          approach for creating a stochastic MDP
%
% Inputs:
%       world:                  A structure containing basic parameters for
%                               the mountain car problem
%       T:                      Transition model with elements initialized
%                               to zero
%       R:                      Expected reward model with elements
%                               initialized to zero
%       num_samples:            Number of samples to use for creating the
%                               stochastic model
%
% Outputs:
%       T:                      Transition model with elements T{a}(s,s')
%                               being the probability of transition to
%                               state s' from state s taking action a
%       R:                      Expected reward model with elements
%                               R{a}(s,s') being the expected reward on
%                               transition from s to s' under action a
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% --
% Revision history
% [20.03.07, SZ]    first version

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);

    % 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);
%             p_k = cur_state(1);
%             v_k = cur_state(2);
%             v_k_next = v_k + 0.001*action_index


            % [TODO] 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

                [next_state,reward,is_goal_state] = one_step_mc_model(world, 
cur_state, action)
                next_state(1) = next_state(1) + normrnd(0,0.001);
                next_state(2) = next_state(2) + normrnd(0,0.005);
                next_state_nearest = nearest_state_index_lookup(STATES, 
next_state);
                T{action_index}(state_index,next_state_nearest) = 
T{action_index}(state_index,next_state_nearest) + 1/num_samples;
                % Update transition and reward models
                % T{action_index}(state_index, next_state_index) = ...;
                R{action_index}(state_index, next_state_nearest) = reward;
            end
        end
    end
end
```

1.2 What is the stochastic element in the modelling process and what is its significance ?

The original system has continuous system dynamics. However it is impractical to assign value function to uncountably infinite number of points. Hence, function approximation methods has to be utilised to compute value function defined over discrete state space. The stochastic MDP formulation helps to build piecewise approximations of the value function.

1.3 What modelling parameters would have the most impact on the quality of the solution ?

The variance of the added noise and its mean has most impact on the quality of the solution. Mean is set to zero, but variance is set to a small number for getting good results. In addition to this, the number discretisation points would also have an impact on the solution.

2.1) main_p2_mc_rl.m

```
% main_p2_mc_rl: Main script for Problem 4.2 mountain car (RL approach)
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
```

```
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% --
% Revision history
% [20.03.07, SZ]    first version

clear all;
close all;
clc;

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

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

%% Problem 4.2 (a)-(b) Create stochastic MDPs for the mountain car problem
% [TODO] Load mountain car model
% change model name correspondingly:
%     (a) 'mountain_car_nn' for the nearest neighbour method
%     (b) 'mountain_car_li' for the linear interpolation approach
load('mountain_car_model/mountain_car_nn');

%% Generalized policy iteration
% Algorithm parameters
precision_pi = 0.1;
precision_pe = 0.01;
max_ite_pi = 100;
max_ite_pe = 10;

% Solve MDP
[v_gpi, policy_gpi] = generalized_policy_iteration_mc(world, precision_pi, ...
    precision_pe, max_ite_pi, max_ite_pe);

% Visualization
plot_value = true;
plot_flowfield = true;
plot_visualize = true;
plot_title = 'Generalized Policy Iteration';
hdl_gpi = visualize_mc_solution(world, v_gpi, policy_gpi, plot_value, ...
    plot_flowfield, plot_visualize, plot_title, save_dir);

% Save results
save(strcat(save_dir, 'gpi_results.mat'), 'v_gpi', 'policy_gpi');
```

generalized_policy_iteration_mc.m

```
% generalized_policy_iteration: Function solving the given MDP using the
%                               Generalized Policy Iteration algorithm
%
% Inputs:
%       world:                  A structure defining the MDP to be solved
%       precision_pi:           Maximum value function change before
%                               terminating Policy Improvement step
%       max_ite_pi:             Maximum number of iterations for Policy
%                               Improvement loop
%       precision_pe:           Maximum value function change before
%                                terminating Policy Evaluation step
%       max_ite_pe:              Maximum number of iterations for Policy
%                                Evaluation loop
%
% Outputs:
%       V:                       An array containing the value at each state
%       policy_index:            An array summarizing the index of the
%                                optimal action index at each state
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.03.07, SZ]    first version

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

    % Discrete states
    POS = world.mdp.POS;
    VEL = world.mdp.VEL;

    % 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., [0,1,0,0,0] 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
```

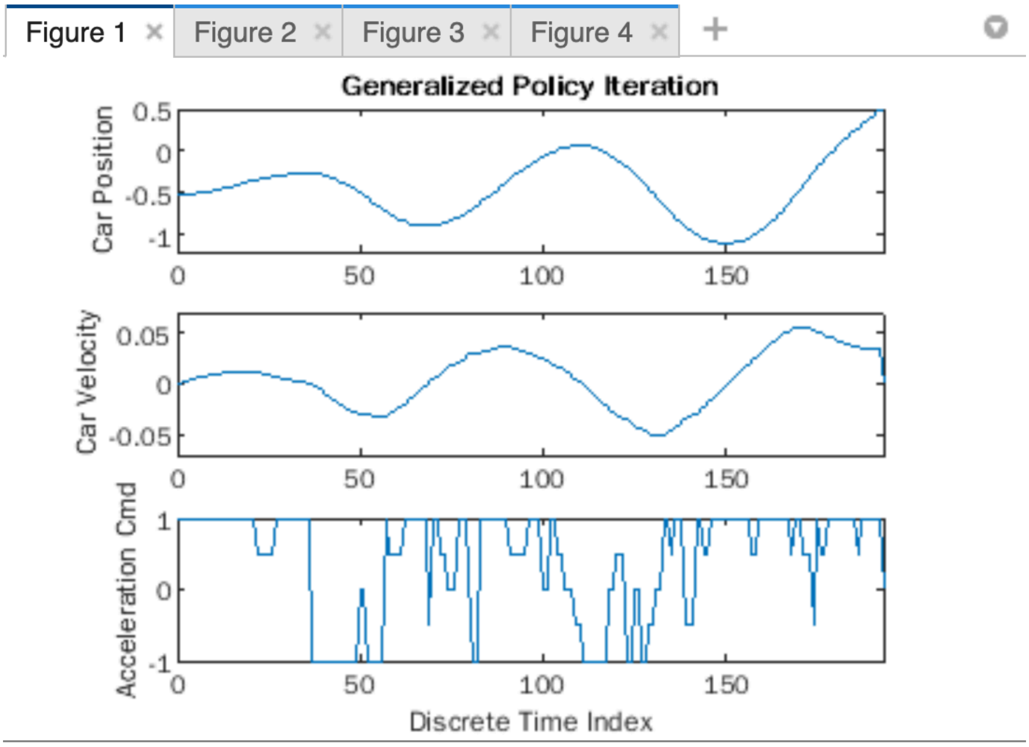
```
        selected_action = random_act_index(s);
        policy(s, selected_action) = 1;
    end
    iterations_pi = 0;
    while iterations_pi <= max_ite_pi
        iterations_pe = 0;
        iterations_pi = iterations_pi + 1;
        %% [TODO] policy Evaluation (PE) (Section 2.6 of [1])
        while iterations_pe <= max_ite_pe
            delta = 0;
            iterations_pe = iterations_pe + 1;
            for s = 1:1:num_states
                v = V(s,1);

                %%%%%%Computation of V%%%%%%%%
                temp_v = 0; %temporary variable for value function computation

                for a = 1:num_actions
                    temp_R = 0; %temporary variable for expected return
computation
                    for s_prime = 1:num_states
                       temp_R = temp_R +  T{a}(s,s_prime)*(R{a}(s,s_prime) +
gamma*V(s_prime,1));
                    end
                 temp_v = temp_v + policy(s,a)*temp_R;

                end
                %finished computation of value function
                 V(s,1) = temp_v;
                abs_diff = abs(v-V(s,1));
                delta = max(abs_diff,delta);
                if delta< precision_pe
                    break
                end
            end

        end
        % V = ...;

        %% [TODO] Policy Improvment (PI) (Section 2.7 of [1])


        policy_is_stable = true;

        for s = 1:num_states
            b = policy(s,:);
            %compute argmax of value function
            for a = 1:num_actions
                temp_R = 0; %temporary variable for expected return computation
                for s_prime = 1:num_states
                       temp_R = temp_R +  T{a}(s,s_prime)*( R{a}(s,s_prime) +
gamma*V(s_prime,1));
                end
                if a==1
                    temp = temp_R;
                    arg_max = a;

                end

                if temp_R >= temp

                    temp = temp_R;
                    arg_max = a;
                end

            end
```

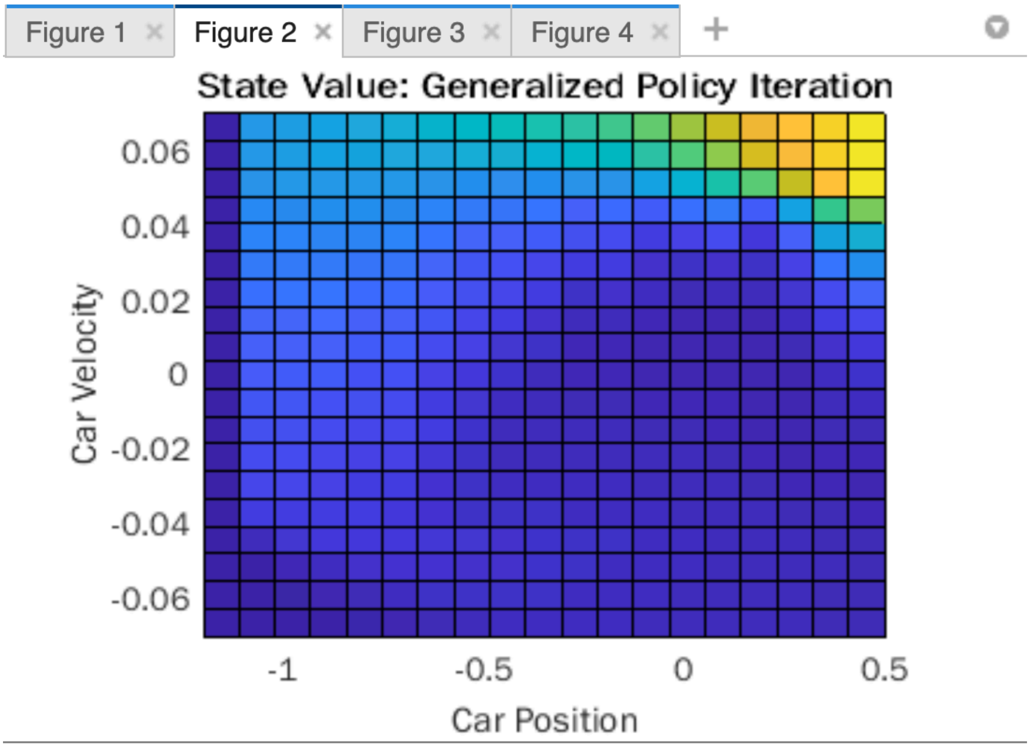
```
             policy(s,:) = zeros(1,num_actions);   %updating policy
             policy(s,arg_max) = 1;

             if find(b) ~= find(policy(s,:))
                fprintf('policy not stable')
                policyISstable = false;
             end

        end


         if policyISstable == true

            break
        end
        % policy = ...;

        % Check algorithm convergence
        % if ...
        %       break
        % end
    end

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

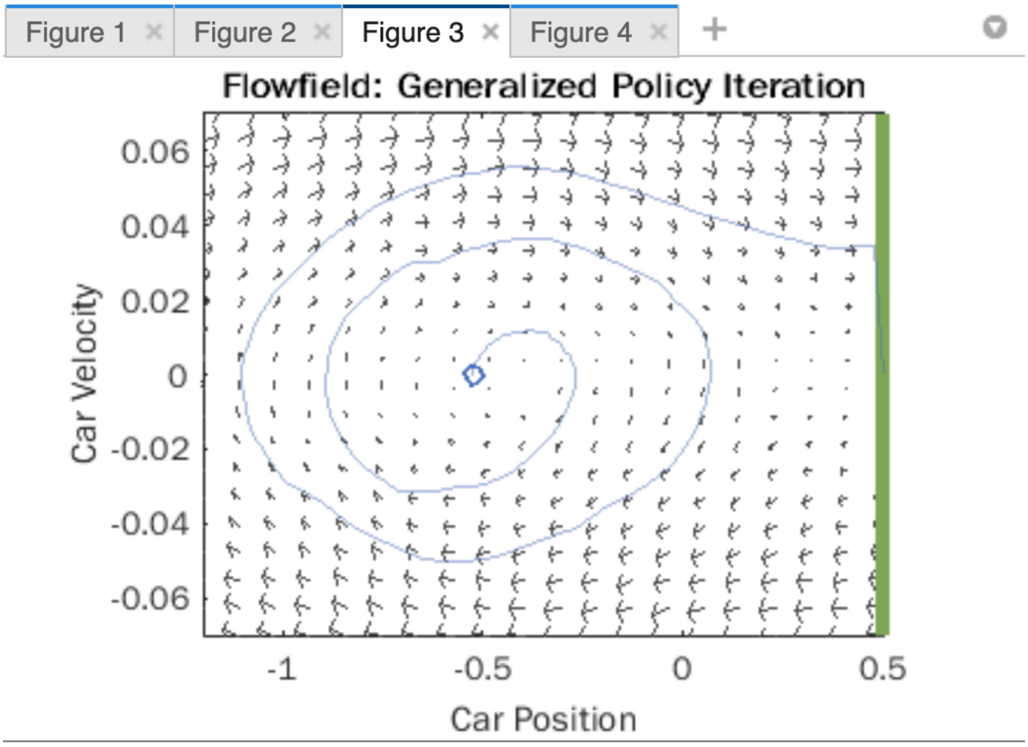
Q: Was the learning algorithm able to find this solution? If not, why do you think that is the case?

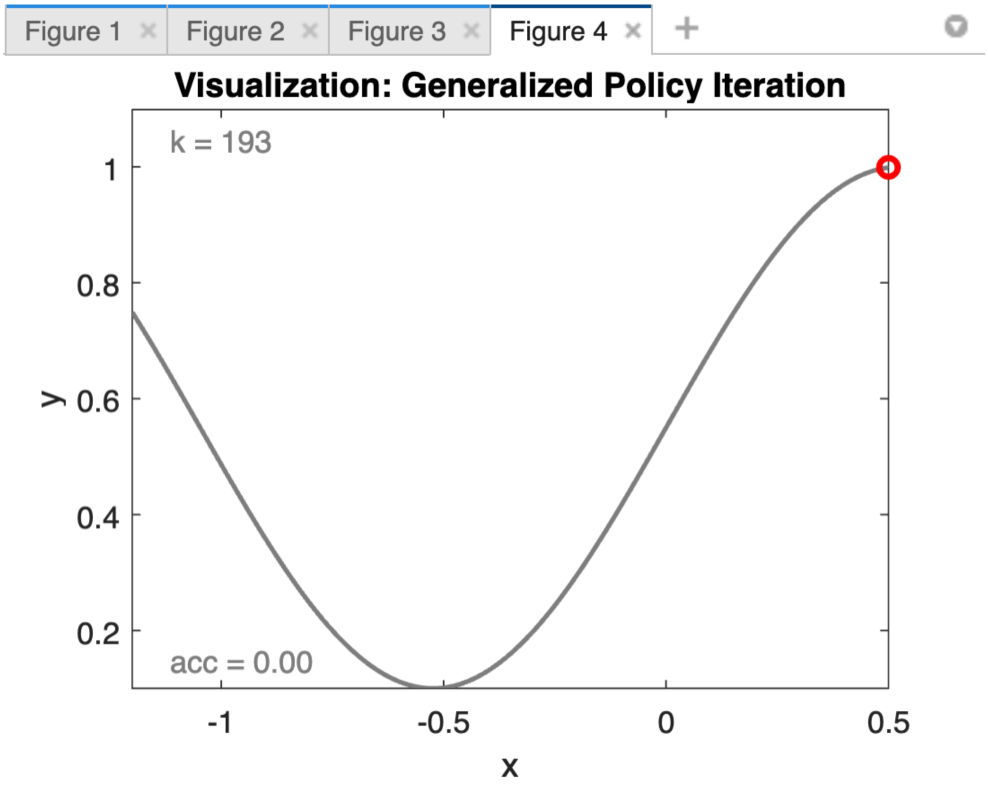
Yes, after tuning the noise parameters.

Converged Heat Map & Results(Nearest Neighbour Case)









build_stochastic_mdp_li.m

```
% build_stochastic_mdp_li: Function implementing the Linear Interpolation
%                          approach for creating a stochastic MDP
%
% Inputs:
%       world:                  A structure containing basic parameters for
%                               the mountain car problem
%       T:                      Transition model with elements initialized
%                               to zero
%       R:                      Expected reward model with elements
%                               initialized to zero
%
% Outputs:
%       T:                      Transition model with elements T{a}(s,s')
%                               being the probability of transition to
%                               state s' from state s taking action a
%       R:                      Expected reward model with elements
%                               R{a}(s,s') being the expected reward on
%                               transition from s to s' under action a
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 4
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% --
% Revision history
% [20.03.07, SZ]    first version

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
```

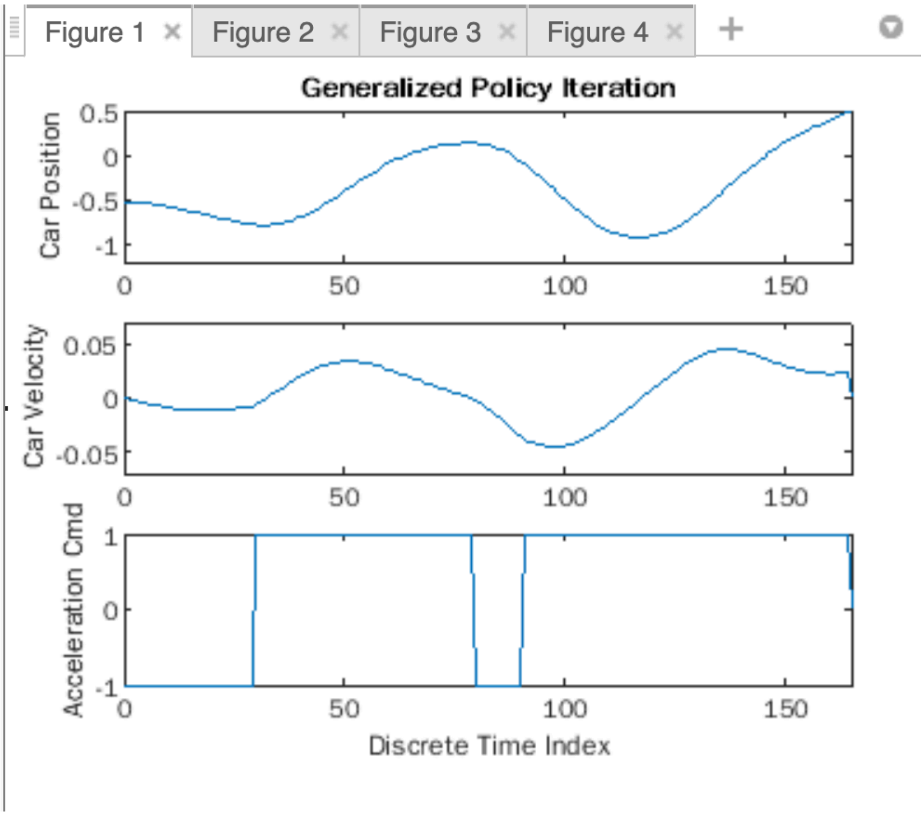


Figure 1
Figure 2
Figure 3
Figure 4
Generalized Policy Iteration
Car Position
0.5
0
-0.5
-1
0
50
100
150
Car Velocity
0.05
0
-0.05
0
50
100
150
Acceleration Cmd
1
0
-1
0
50
100
150
Discrete Time Index

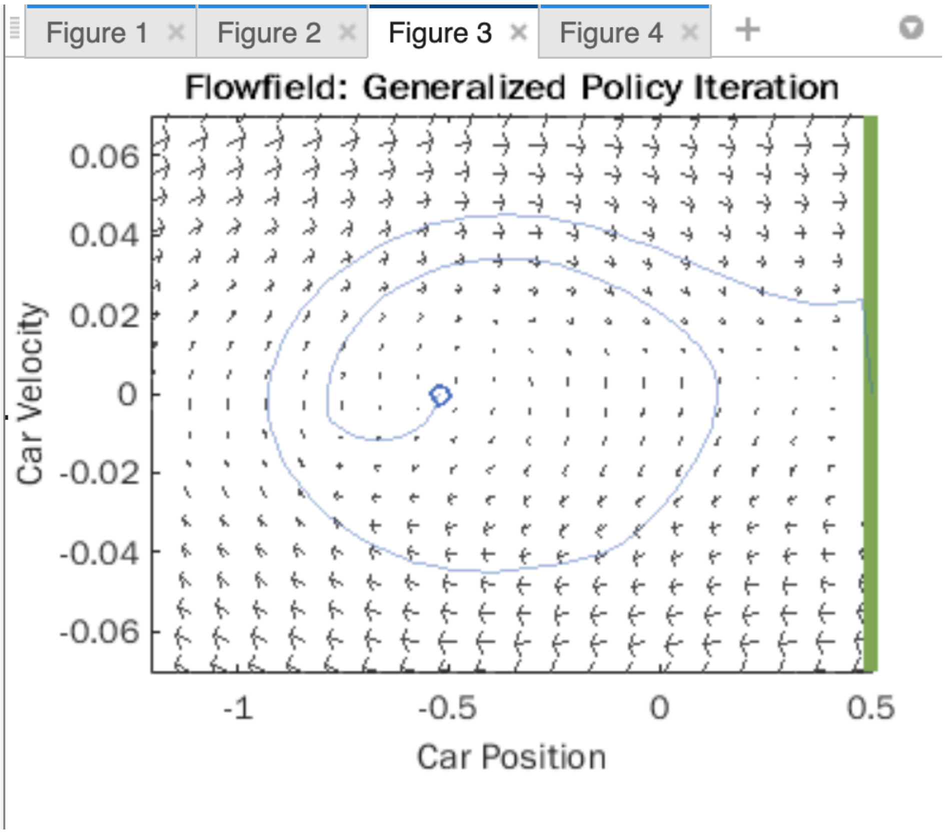


Figure 1
Figure 2
Figure 3
Figure 4
Flowfield: Generalized Policy Iteration
Car Velocity
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-1
-0.5
0
0.5
Car Position

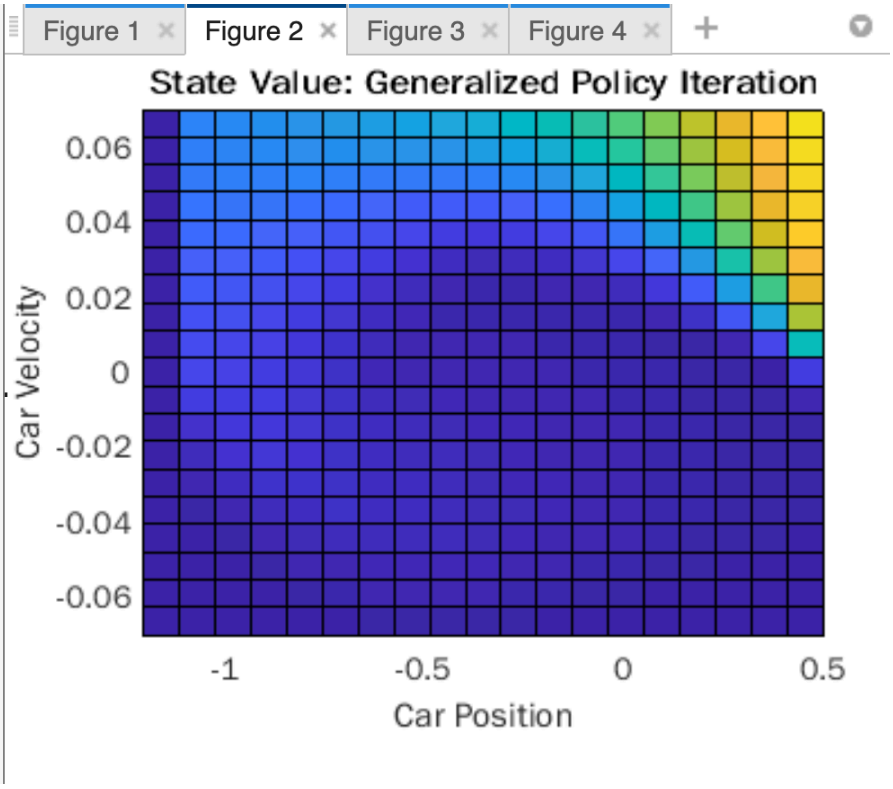


Figure 1
Figure 2
Figure 3
Figure 4
State Value: Generalized Policy Iteration
Car Velocity
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-1
-0.5
0
0.5
Car Position

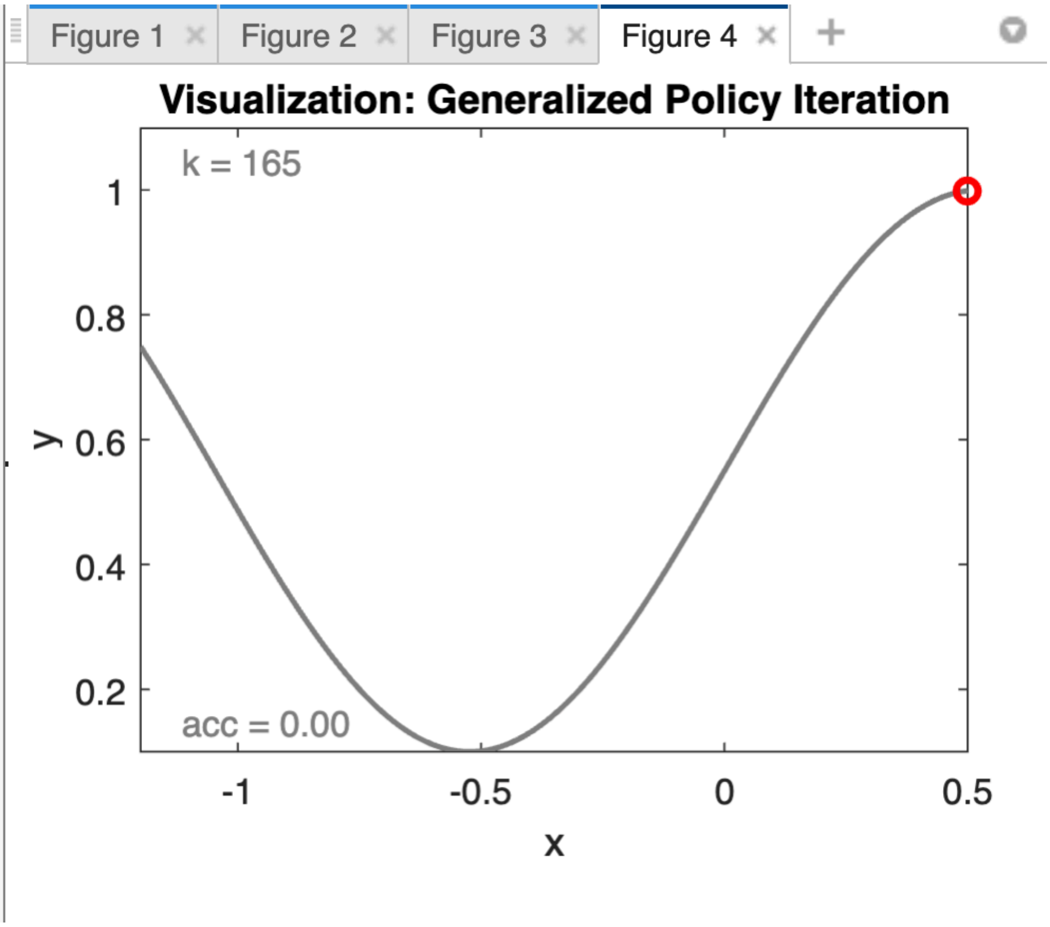


Figure 1
Figure 2
Figure 3
Figure 4
Visualization: Generalized Policy Iteration
k = 165
acc = 0.00
y
1
0.8
0.6
0.4
0.2
-1
-0.5
0
0.5
x

Effectiveness of LI and NN:

It is observed that the linear interpolation method is more effective than nearest neighbours method. For NN, the noise parameters has to be carefully tuned to get good results. However, the linear interpolation method is much simpler to implement, as it does not contain any hyper parameters. Both methods can be used to solve the mountain car problem effectively.

APPENDIX: (Some helper functions)

1. sample_from_epsilon_policy.m - function to sample from epsilon soft policy.

```
function action_index = sample_from_epsilon_policy(epsilon,policy)

val = rand();

%find greedy action

for a = 1:4

    if a == 1

     temp = policy(a);

    end

   if policy(a) >= temp
        temp = policy(a);
        greedy_action = a;

   end

end


if val <= 1-epsilon

    action_index = greedy_action;

elseif val > 1- epsilon

    while true
     action_non_greedy_index = randi([1,4]);

     if action_non_greedy_index ~= greedy_action
        action_index = action_non_greedy_index;
      break

     end
    end
    %select non greedy action
end



end
```

2. initialize_random_policy.m - function to initialise random epsilon soft policy

```
function policy = initialize_random_policy(epsilon,n_states,n_actions)
    % Dimensions
   policy = zeros(n_states,n_actions);
   min_prob = epsilon/n_actions;
   max_prob = 1 - epsilon*(1-(1/n_actions));
   policy(1:n_states,1:n_actions) = min_prob;

   for i = 1:n_states
       greedy_index = randi([1,4]);
       policy(i,greedy_index) = max_prob;

   end


end
```

3. arg_max_Q.m - function to comute argmax(Q(x,a))

```
function u = arg_max_Q(Q,x)

for a = 1:4

    if a == 1
     temp = Q(x,a);

    end
  if Q(x,a) >= temp

      temp = Q(x,a);
      a_max =a;

  end

end
u = a_max;
end

%find greedy action
```