- (A)

```
%%%%%%% CS 229 Machine Learning %%%%%%%%%%%
%%%%%%% Programming Assignment 4 %%%%%%%%%%
%%%
%%% Parts of the code (cart and pole dynamics, and the state
%%% discretization) are adapted from code available at the RL repository
%%% http://www-anw.cs.umass.edu/rlr/domains.html
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This file controls the pole-balancing simulation. You need to write
% code in places marked "CODE HERE" only.

% Briefly, the main simulation loop in this file calls cart_pole.m for
% simulating the pole dynamics, get_state.m for discretizing the
% otherwise continuous state space in discrete states, and show_cart.m
% for display.

% Some useful parameters are listed below.

% NUM_STATES: Number of states in the discretized state space
% You must assume that states are numbered 1 through NUM_STATES. The
% state numbered NUM_STATES (the last one) is a special state that marks
% the state when the pole has been judged to have fallen (or when the
% cart is out of bounds). However, you should NOT treat this state any
% differently in your code. Any distinctions you need to make between
% states should come automatically from your learning algorithm.

% After each simulation cycle, you are supposed to update the transition
% counts and rewards observed. However, you should not change either
% your value function or the transition probability matrix at each
% cycle.

% Whenever the pole falls, a section of your code below will be
% executed. At this point, you must use the transition counts and reward
% observations that you have gathered to generate a new model for the MDP
% (i.e., transition probabilities and state rewards). After that, you
% must use value iteration to get the optimal value function for this MDP
% model.

% TOLERANCE: Controls the convergence criteria for each value iteration
% run
% In the value iteration, you can assume convergence when the maximum
% absolute change in the value function at any state in an iteration
% becomes lower than TOLERANCE.

% You need to write code that chooses the best action according
% to your current value function, and the current model of the MDP. The
% action must be either 1 or 2 (corresponding to possible directions of
% pushing the cart).

% Finally, we assume that the simulation has converged when
% 'NO_LEARNING_THRESHOLD' consecutive value function computations all
% converged within one value function iteration. Intuitively, it seems
% like there will be little learning after this, so we end the simulation
% here, and say the overall algorithm has converged.
```

```
% Learning curves can be generated by calling plot_learning_curve.m (it
% assumes that the learning was just executed, and the array
% time_steps_to_failure that records the time for which the pole was
% balanced before each failure are in memory). num_failures is a variable
% that stores the number of failures (pole drops / cart out of bounds)
% till now.

% Other parameters in the code are described below:

% GAMMA: Discount factor to be used

% The following parameters control the simulation display; you dont
% really need to know about them:

% pause_time: Controls the pause between successive frames of the
% display. Higher values make your simulation slower.
% min_trial_length_to_start_display: Allows you to start the display only
% after the pole has been successfully balanced for at least this many
% trials. Setting this to zero starts the display immediately. Choosing a
% reasonably high value (around 100) can allow you to rush through the
% initial learning quickly, and start the display only after the
% performance is reasonable.

%%%%%%%%%% Simulation parameters %%%%%%%%%%
rng(1)

pause_time = 0.001;
min_trial_length_to_start_display = 100;
display_started=0;

NUM_STATES = 163;

GAMMA=0.9995;

TOLERANCE=0.01;

NO_LEARNING_THRESHOLD = 20;

%%%%%%%%%%  End parameter list   %%%%%%%%%%

% Time cycle of the simulation
time=0;

% These variables perform bookkeeping (how many cycles was the pole
% balanced for before it fell). Useful for plotting learning curves.
time_steps_to_failure=[];
num_failures=0;
time_at_start_of_current_trial=0;

max_failures=500; % You should reach convergence well before this.

% Starting state is (0 0 0 0)
% x, x_dot, theta, theta_dot represents the actual continuous state vector
x = 0.0; x_dot = 0.0; theta = 0.0; theta_dot = 0.0;

% state is the number given to this state - you only need to consider
% this representation of the state
state = get_state(x, x_dot, theta, theta_dot);

if min_trial_length_to_start_display==0 || display_started==1
```

```
  show_cart(x, x_dot, theta, theta_dot, pause_time);
end

%%% CODE HERE: Perform all your initializations here %%%

% Assume no transitions or rewards have been observed
% Initialize the value function array to small random values (0 to 0.10,
% say)

V = rand(NUM_STATES, 1) .* 0.001;

% Initialize the transition probabilities uniformly (ie, probability of
% transitioning for state x to state y using action a is exactly
% 1/NUM_STATES).
P_action_1 = ones(NUM_STATES, NUM_STATES) ./ NUM_STATES;
P_action_2 = ones(NUM_STATES, NUM_STATES) ./ NUM_STATES;

%Initialize all state rewards to zero.
Reward = zeros(NUM_STATES, 1);

%
P_record_1 = zeros(NUM_STATES, NUM_STATES);
P_record_2 = zeros(NUM_STATES, NUM_STATES);

R_record = zeros(NUM_STATES, 1);
R_accumu = zeros(NUM_STATES, 1);

consecutive = 0;
%%%% END YOUR CODE %%%%%%%%%%%%%%%%%%%%%%%%%%%%




%%% CODE HERE (while loop condition) %%%
% This is the criterion to end the simulation
% You should change it to terminate when the previous
% 'NO_LEARNING_THRESHOLD' consecutive value function computations all
% converged within one value function iteration. Intuitively, it seems
% like there will be little learning after this, so end the simulation
% here, and say the overall algorithm has converged.

%while num_failures<max_failures
while (num_failures<max_failures && consecutive < NO_LEARNING_THRESHOLD)

  %%% CODE HERE: Write code to choose action (1 or 2) %%%
  max_1 = P_action_1(state, :) * V;
  max_2 = P_action_2(state, :) * V;

  if (max_1 > max_2)
      action = 1;

  elseif (max_1 < max_2)
      action = 2;

  elseif rand(1) > 0.5
      action = 1;
  else
      action = 2;
  end
```

```
  %%% END YOUR CODE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  % Get the next state by simulating the dynamics
  [x, x_dot, theta, theta_dot] = cart_pole(action, x, x_dot, theta, theta_dot);

  % Increment simulation time
  time = time + 1;

  % Get the state number corresponding to new state vector
  new_state = get_state(x, x_dot, theta, theta_dot);

  if display_started==1
    show_cart(x, x_dot, theta, theta_dot, pause_time);
  end

  % Reward function to use - do not change this!
  if (new_state==NUM_STATES)
    R=-1;
  else
    %R=-abs(theta)/2.0;
    R=0;
  end

  %%% CODE HERE: Perform updates %%%%%%%%%

  % A transition from 'state' to 'new_state' has just been made using
  % 'action'. The reward observed in 'new_state' (note) is 'R'.
  % Write code to update your statistics about the MDP - i.e., the
  % information you are storing on the transitions and on the rewards
  % observed. Do not change the actual MDP parameters, except when the
  % pole falls (the next if block)!

  if action == 1
      P_record_1(state, new_state) = P_record_1(state, new_state) + 1;
  else
      P_record_2(state, new_state) = P_record_2(state, new_state) + 1;
  end

  R_record(new_state) = R_record(new_state) + 1;
  R_accumu(new_state) = R_accumu(new_state) + R;

  % Recompute MDP model whenever pole falls
  % Compute the value function V for the new model
  if (new_state==NUM_STATES)

    % Update MDP model using the current accumulated statistics about the
    % MDP - transitions and rewards.
    % Make sure you account for the case when total_count is 0, i.e., a
    % state-action pair has never been tried before, or the state has
    % never been visited before. In that case, you must not change that
    % component (and thus keep it at the initialized uniform distribution).
    % update transition matrix and reward at the observed states
    p_index_1 = max(P_record_1,[], 2) > 0;
    p_index_2 = max(P_record_2,[], 2) > 0;

    P_action_1(p_index_1, :) = P_record_1(p_index_1, :) ./ ...
        sum(P_record_1(p_index_1, :), 2);
    P_action_2(p_index_2, :) = P_record_2(p_index_2, :) ./ ...
        sum(P_record_2(p_index_2, :), 2);
```

```
    index = R_record > 0;
    Reward(index) = R_accumu(index) ./ R_record(index);

    % update Value by value-iteraion
    count = 0;
    P1 = P_action_1;
    P2 = P_action_2;
    while true
        count = count + 1;
        V_old = V;
        max_1 = P1 * V;
        max_2 = P2 * V;
        V = Reward + GAMMA * max(max_1, max_2);
        % Check convergence
        if sum((V - V_old).^2) < TOLERANCE
            break
        end
    end

    % If the iteration only continues once
    if count == 1
        consecutive = consecutive + 1;
    else
        consecutive = 0;
    end

    % Perform value iteration using the new estimated model for the MDP
    % The convergence criterion should be based on TOLERANCE as described
    % at the top of the file.
    % If it converges within one iteration, you may want to update your
    % variable that checks when the whole simulation must end


    %pause(0.2); % You can use this to stop for a while!

  end


  %%% END YOUR CODE %%%%%%%%%%%%%%%%%%%


  % Dont change this code: Controls the simulation, and handles the case
  % when the pole fell and the state must be reinitialized

  if (new_state == NUM_STATES)
    num_failures = num_failures+1;
    time_steps_to_failure(num_failures) = time - time_at_start_of_current_trial;
    time_at_start_of_current_trial = time;

    time_steps_to_failure(num_failures)

    if (time_steps_to_failure(num_failures)> ...
 min_trial_length_to_start_display)
      display_started=1;
    end

    % Reinitialize state
    x = -1.1 + rand(1)*2.2;
    %x=0.0;
    x_dot = 0.0; theta = 0.0; theta_dot = 0.0;
    state = get_state(x, x_dot, theta, theta_dot);
```

```
    else
      state=new_state;
    end
  end

  % Plot the learning curve (time balanced vs trial)
  plot_learning_curve
```

The algorithm took 120 trials to stop itself. But from the plot below, we could find that it approximately converged after 40 trials.

- (B)